Quartus II Version 6.0 Handbook

Volume 4: SOPC Builder

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com

QII5v4-6.0

Contents

Chapter Revision Dates

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

Chapter 1.	Introduction to SOPC Builder	
	Revised:	*May 2006*
	Part number:	*QII54001-6.0.0*
Chapter 2.	Tour of the SOPC Builder User Interface	
	Revised:	*May 2006*
	Part number:	*QII54002-6.0.0*
Chapter 3.	Avalon Switch Fabric	
	Revised:	*May 2006*
	Part number:	*QII54003-6.0.0*
Chapter 4.	SOPC Builder Components	
	Revised:	*May 2006*
	Part number:	*QII54004-6.0.0*
Chapter 5.	Component Editor	
	Revised:	*May 2006*
	Part number:	*QII54005-6.0.0*
Chapter 6.	Archiving SOPC Builder Projects	
	Revised:	*May 2006*
	Part number:	*QII54017-6.0.0*
Chapter 7.	Board Description Editor	
	Revised:	*May 2006*
	Part number:	*QII54017-6.0.0*
Chapter 8.	Pin Mapper	
	Revised:	*May 2006*
	Part number:	*QII54016-6.0.0*
Chapter 9.	Building Memory Subsystems Using SOPC Builder	
	Revised:	*May 2006*
	Part number:	*QII54006-6.0.0*

Chapter 10.	Developing Components for SOPC Builder	
	Revised:	*May 2006*
	Part number:	*QII54007-6.0.0*
Chapter 11.	Building Systems with Multiple Clock Domains	
	Revised:	*May 2006*
	Part number:	*QII54008-6.0.0*

About this Handbook

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

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/	www.altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	+1 408-544-8767 7:00 a.m. to 5:00 p.m. (GMT -8:00) Pacific Time
Product literature	www.altera.com	www.altera.com
Altera literature services	literature@altera.com	literature@altera.com
Non-technical customer service	(800) 767-3753	+ 1 408-544-7000 7:00 a.m. to 5:00 p.m. (GMT -8:00) Pacific Time
FTP site	ftp.altera.com	ftp.altera.com

Typographic Conventions

This document uses the typographic conventions shown below.

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

Section I. SOPC Builder Features

Section I of this volume introduces the SOPC Builder system integration tool, and describes the main features. Chapters in this section serve to answer the following questions:

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

This section includes the following chapters:

Revision History

The following table shows the revision history for Chapters 1–8.

Chapter(s)	Date / Version	Changes Made
1	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.
2	May 2006, v6.0.0	No change from previous release.
	October 2005, v5.1.0	● Updated for the Quartus II software version 5.1. ● Added pipeline for high performance details. ● Added endian conversion details.
	May 2005, v5.0.0	No change from previous release.
3	May 2006, v6.0.0	No change from previous release.
	October 2005, v5.1.0	Updated for the Quartus II software version 5.1.
	August 2005, v5.0.1	● Added burst transfer management details. ● Updated pipeline management details.
	May 2005, v5.0.0	No change from previous release.
	February 2005, v1.0	Initial release.

Chapter(s)	Date / Version	Changes Made
4	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.
5	May 2006, v6.0.0	No change from previous release.
	December 2005, v5.1.1	● Added section “Naming Signals for Automatic Type and Interface Recognition.” ● Added section “Templates for Interfaces to External Logic.” ● Clarified operation of the Save command. ● Updated all screenshots.
	October 2005, v5.1.0	No change from previous release.
	May 2005, v5.0.0	No change from previous release.
6	May 2006, v6.0.0	Initial release.
7	May 2006, v6.0.0	Chapter 7 was previously chapter 6. No change to content.
	October 2005 v5.1.0	Initial release.
8	May 2006, v6.0.0	Chapter 8 was previously chapter 7. No change to content.
	October 2005 v5.1.0	Initial release.

1. Introduction to SOPC Builder

QII54001-6.0.0

Overview

SOPC Builder is a powerful system development tool for creating systems based on 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 and is available to all Altera® customers.

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 arbitrary SOPC designs 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 had to manually write top-level HDL files that wire together the pieces of the system. Using SOPC Builder, you specify the system components in a graphical user interface (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.

In addition to its role as a hardware generation tool, SOPC Builder also serves as the starting point for system simulation and embedded software creation. SOPC Builder provides features to ease writing software and to accelerate system simulation.

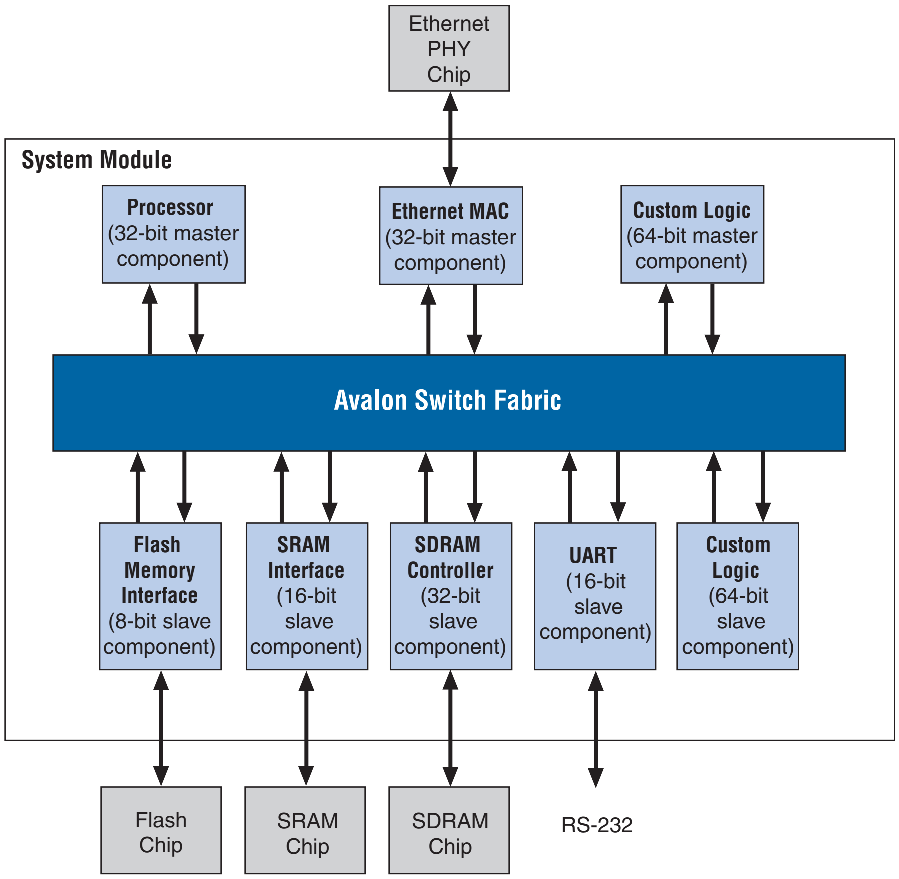
This chapter introduces you to the architectural structure of systems built with SOPC Builder, and describes the primary functions of SOPC Builder.

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. SOPC Builder connects multiple components together to create a top-level HDL file called the *system module*. SOPC Builder generates *Avalon® switch fabric* that contains logic to manage the connectivity of all components in the system. Figure 1–1 shows an example of a multi-master system module with Avalon switch fabric connecting the multiple master and slave components.

Figure 1–1. Example of a System Module Generated by SOPC Builder


Ethernet PHY Chip
System Module
Processor (32-bit master component)
Ethernet MAC (32-bit master component)
Custom Logic (64-bit master component)
Avalon Switch Fabric
Flash Memory Interface (8-bit slave component)
SRAM Interface (16-bit slave component)
SDRAM Controller (32-bit slave component)
UART (16-bit slave component)
Custom Logic (64-bit slave component)
Flash Chip
SRAM Chip
SDRAM Chip
RS-232

SOPC Builder Components

SOPC Builder components are the building blocks of the system module. SOPC Builder components use the Avalon interface 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. The Avalon interface uses an address-mapped read/write protocol that allows master components to read and/or write any slave component.

For details on the Avalon interface, see the *Avalon Interface Specification* at **www.altera.com**.

A component can be a logical device that is entirely contained within the system module, such as the processor component in Figure 1–1 on page 1–2. Alternately, a component can act as an interface to an off-chip device, such as the SRAM interface component in Figure 1–1 on page 1–2. 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 Ethernet MAC in Figure 1–1 on page 1–2.

A component can be instantiated more than once per design.

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

- Microprocessors, such as the Nios II processor
- Microcontroller peripherals
- 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
- Interfaces to off-chip devices
  - Memory controllers
  - Buses and bridges
  - Application-specific standard products (ASSP)
  - Application-specific integrated circuits (ASIC)
  - Processors

SOPC Builder Ready Components

Altera awards the SOPC Builder Ready certification to intellectual property (IP) designs that have plug–and–play integration with SOPC Builder. These functions may be accompanied by software drivers, low-level routines, or other software design files.

Altera's OpenCore® and OpenCore Plus evaluation programs allow you to "test drive" an SOPC Builder component both in simulation and in hardware before you buy. You can download evaluations of Altera IP functions directly from **www.altera.com/IPMegastore**. For IP functions provided by third-party vendors, contact the vendor directly to obtain an OpenCore evaluation.

Check the Altera web site at **www.altera.com** for up–to–date information about available SOPC Builder Ready components. You can identify SOPC Builder Ready components by the logo shown in Figure 1–2.

Figure 1–2. The SOPC Builder Ready Certification Logo

User-Defined Components

SOPC Builder provides an easy method for you to develop and connect your own components. With the Avalon interface, user-defined logic need only adhere to a simple interface based on address, data, read-enable, and write-enable signals.

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

1. Define the interface to the user-defined component.

2. If the component logic resides on-chip, 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 Ready 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, see the *Developing SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For complete detail on the file structure of a component, see the *SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For details on the SOPC Builder component editor, see the *Component Editor* chapter in Volume 4 of the *Quartus II Handbook*.

Avalon Switch Fabric

The Avalon switch fabric is the glue that binds SOPC Builder-generated systems together. The Avalon switch 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. SOPC Builder generates the Avalon switch fabric automatically, so that you do not have to manually perform the tedious, error-prone task of connecting hardware modules.

The purpose of SOPC Builder is to abstract away the complexity of interconnect logic, allowing designers to focus on the details of their custom components and the high-level system architecture. Automatically generating the Avalon switch fabric is the keystone to achieving this purpose. Avalon switch fabric in the system module is like air for humans: Its existence is essential, but largely ignored. Because SOPC Builder generates Avalon switch fabric automatically, most users do not interact directly with it or the HDL that describes it.

For further details, see the *Avalon Switch Fabric* chapter in Volume 4 of the *Quartus II Handbook*.

Functions of SOPC Builder

This section describes the fundamental functions of SOPC Builder.

Defining & Generating the System Hardware

The purpose of the SOPC Builder GUI is to allow you to easily define the structure of a hardware system, and then generate the system. The GUI is designed for the tasks of adding components to a system, configuring the components, and specifying how they connect together.

After you add all components and specify all necessary system parameters, SOPC Builder is ready to generate the Avalon switch fabric and output the HDL files that describe the system. 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**)
- (Optional) Software files for embedded software development, such as a memory-map header file and component drivers
- (Optional) Testbench for the system module and ModelSim® simulation project files

After you generate the system module, it can be compiled directly by the Quartus II software, or instantiated in a larger FPGA design.

For more detail on the SOPC Builder GUI for defining and generating systems, see the *Tour of the SOPC Builder User Interface* chapter in Volume 4 of the *Quartus II Handbook*.

Creating a Memory Map for Software Development

For each microprocessor in the system, SOPC Builder optionally generates a header file that defines the address of each slave component. In addition, each slave component can provide software drivers and other software functions and libraries for the processor.

The process for writing 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 & Testbench

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), Evaluation Edition. You can download this design suite for free from the Altera Download Center at **www.altera.com**.

2. Tour of the SOPC Builder User Interface

QII54002-6.0.0

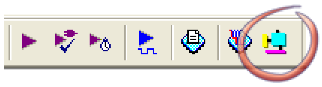
This chapter provides reference on how to access the features available in the SOPC Builder graphical user interface (GUI). This chapter will familiarize you with the main features of the SOPC Builder GUI.

☞ Due to evolution and improvement of the software, the figures in this chapter may not match exactly what you see in the software.

Starting SOPC Builder

Each SOPC Builder system is associated with one Quartus® II project. Therefore, to launch SOPC Builder, you must first open a project in the Quartus II software. With a Quartus II project open, you can launch SOPC Builder by choosing **SOPC Builder** (Tools menu) or by clicking the **SOPC Builder** toolbar button. See Figure 2–1.

Figure 2–1. SOPC Builder Toolbar Button



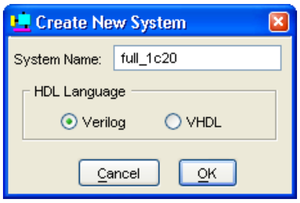
☞ The SOPC Builder toolbar button might not appear by default. To enable it, use the **Customize** window (Tools menu).

Starting a New SOPC Builder System

If an SOPC Builder system does not exist in the current Quartus II project directory, SOPC Builder will display the **Create New System** dialog as shown in Figure 2–2 on page 2–2, and prompt you to specify the following:

- A name for the new SOPC Builder system – This serves as the name of the system module that SOPC Builder will generate.
- Target HDL – This setting determines the output language of the system module.

Figure 2–2. Create New System Dialog



System Name: full_1c20

HDL Language: [✓ Verilog] VHDL

Working with SOPC Builder Systems

SOPC Builder operates on exactly one system at a time, and the system must be associated with a Quartus II project. A Quartus II project may have multiple associated SOPC Builder systems, but typically will have only one (if any).

CAUTION If you integrate multiple SOPC Builder system modules in one Quartus II project, you must make sure all components are named uniquely across all system modules. Otherwise, filename collision will occur.

In SOPC Builder, you can create a new system by choosing **New System** (File menu). You can switch to a different system by choosing **Open System** (File menu).

SOPC Builder saves files in the same directory as the Quartus II project. Each SOPC Builder system is represented by a file named <*system module name*>**.ptf**, which is a plain-text file describing the structure of the system and other system-specific details. In a purely mechanical sense, the SOPC Builder GUI is a **.ptf** file editor.

Changes you make in the SOPC Builder GUI are saved immediately to the **.ptf** file. When you open a system, SOPC Builder creates a back-up file named <*system module name*>**.ptf.bak**, in case you need to revert changes.

System Contents Tab

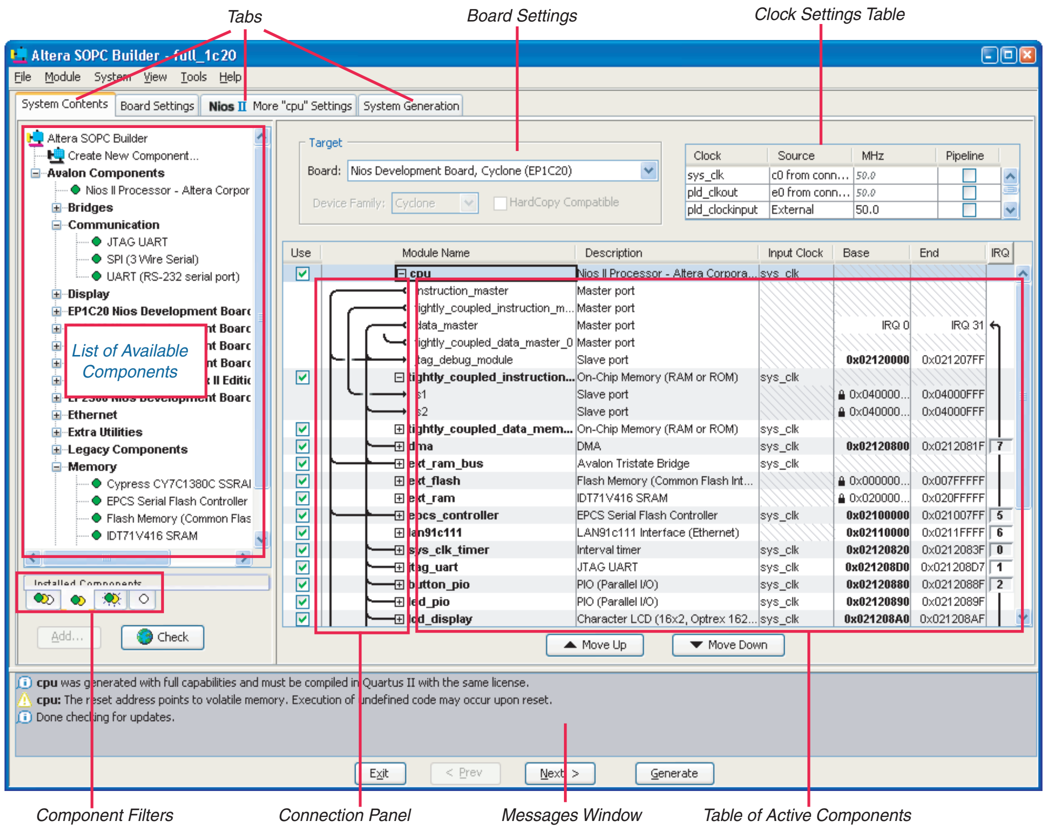
SOPC Builder employs a tabbed user interface. Tasks are categorized by function, and related tasks are presented on the same tab.

The **System Contents** tab is displayed when you open SOPC Builder. It is the view of SOPC Builder that you will use most often. You use the **System Contents** tab to do the following:

- Add components to a system
- Configure the components
- Specify connections between components

Figure 2–3 lists the elements of the **System Contents** tab. See Table 2–1 on page 2–4 for details.

Figure 2–3. Elements of the System Contents Tab



Altera SOPC Builder
- Create New Component...
- **Avalon Components**
  - Nios II Processor - Altera Corpor
  - **Bridges**
  - **Communication**
    - JTAG UART
    - SPI (3 Wire Serial)
    - UART (RS-232 serial port)
  - **Display**
  - **EP1C20 Nios Development Boarc**
  - **nt Boarc**
  - **nt Boarc**
  - **nt Boarc**
  - **II Editic**
  - **nt Boarc**
  - **Ethernet**
  - **Extra Utilities**
  - **Legacy Components**
  - **Memory**
    - Cypress CY7C1380C SSRAI
    - EPCS Serial Flash Controller
    - Flash Memory (Common Flas
    - IDT71V416 SRAM

Target

Board: Nios Development Board, Cyclone (EP1C20)

Device Family: Cyclone   HardCopy Compatible

Clock	Source	MHz	Pipeline
sys_clk	c0 from conn...	50.0	☐
pld_clkout	e0 from conn...	50.0	☐
pld_clockinput	External	50.0	☐

Use	Module Name	Description	Input Clock	Base	End	IRQ
✓	**cpu**	Nios II Processor - Altera Corpora...	sys_clk			
	instruction_master	Master port				
	tightly_coupled_instruction_m...	Master port				
	data_master	Master port		IRQ 0	IRQ 31	
	tightly_coupled_data_master_0	Master port				
	jtag_debug_module	Slave port		**0x02120000**	0x021207FF	
✓	**tightly_coupled_instruction...**	On-Chip Memory (RAM or ROM)	sys_clk			
	s1	Slave port		🔒 0x040000...	0x04000FFF	
	s2	Slave port		🔒 0x040000...	0x04000FFF	
✓	**tightly_coupled_data_mem...**	On-Chip Memory (RAM or ROM)	sys_clk			
✓	**dma**	DMA	sys_clk	**0x02120800**	0x0212081F	7
✓	**ext_ram_bus**	Avalon Tristate Bridge	sys_clk			
✓	**ext_flash**	Flash Memory (Common Flash Int...		🔒 0x000000...	0x007FFFFF	
✓	**ext_ram**	IDT71V416 SRAM		🔒 0x020000...	0x020FFFFF	
✓	**epcs_controller**	EPCS Serial Flash Controller	sys_clk	**0x02100000**	0x021007FF	5
✓	**lan91c111**	LAN91c111 Interface (Ethernet)		**0x02110000**	0x0211FFFF	6
✓	**sys_clk_timer**	Interval timer	sys_clk	**0x02120820**	0x0212083F	0
✓	**jtag_uart**	JTAG UART	sys_clk	**0x021208D0**	0x021208D7	1
✓	**button_pio**	PIO (Parallel I/O)	sys_clk	**0x02120880**	0x0212088F	2
✓	**led_pio**	PIO (Parallel I/O)	sys_clk	**0x02120890**	0x0212089F	
✓	**lcd_display**	Character LCD (16x2, Optrex 162...	sys_clk	**0x021208A0**	0x021208AF	

cpu was generated with full capabilities and must be compiled in Quartus II with the same license.
cpu: The reset address points to volatile memory. Execution of undefined code may occur upon reset.
Done checking for updates.

Table 2–1 describes the GUI elements shown in Figure 2–3.

Table 2–1. GUI Elements on the System Generation Tab

GUI Element	Description
Tabs	Categorizes GUI controls, based on task.
List of Available Components	Lists the library of available SOPC Builder components, organized by category. Each component appears with a colored dot next to its name. The colors of the dots have the following meanings: ● Green dot – A full, licensed version of the component is installed. ● Yellow dot – A limited, evaluation version of the component is installed. ● White dot – This component is available from Altera or a partner vendor, but is not installed.
Component Filters	Filters which type of components appear in the list of available components.
Table of Active Components *(1)*	Lists the components instantiated in the current system, and allows you to specify the following: ● Name for each component instance ● Base address for each slave port ● Clock source for each component instance ● Interrupt priority (if any) for each slave port
Connection Panel *(1)*	Represents connections between the components, and allows you to perform the following: ● Specify connections between master ports and slave ports. ● Specify arbitration shares for slave ports that are shared by multiple master ports.
Board Settings	Allows you to specify details regarding the target hardware platform: ● Board - If the target board is known and you have an SOPC Builder board description, you can specify the target board. The board setting provides SOPC Builder with information about the target hardware, such as pin-outs and connections to off-chip devices. ● Device Family - If the target board is not known, you can specify a particular device family, which affects the behavior of certain components. ● HardCopy Compatible - Certain components have limited functionality when targeting Altera HardCopy devices. This option allows you to use only components and settings that are compatible with HardCopy devices.
Clock Settings Table	Allows you to define the clock signals used in the system module. For each clock in the clock settings table, you can specify: ● Name ● Source - Clock signals can be provided from an external source, or can be generated by a component inside the system module. ● Frequency in MHz ● Pipeline for high performance - SOPC Builder can pipeline the Avalon switch fabric for the clock domain, which improves f_{MAX} performance.
Messages Window	Displays information, warning, or error messages related to the current system.

Note for Table 2–1:
(1) Options available in the View menu alter how this element is displayed.

You can connect any master port to any slave port, as long as they use the same interface protocol. If they use different interface protocols, they must communicate through a bridge component, such as the AHB–to–Avalon® bridge provided with SOPC Builder.

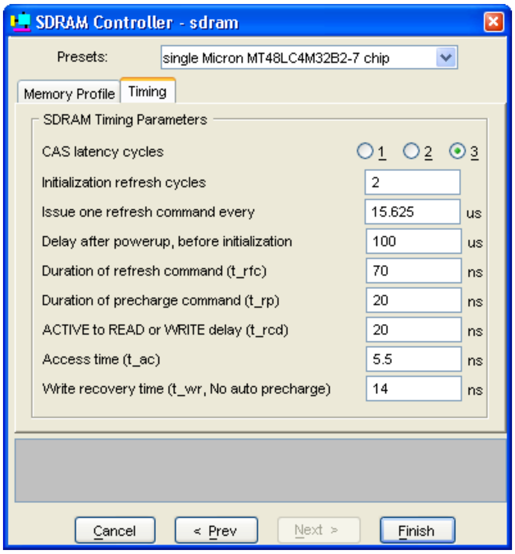
Adding a Component to the System

To add a component to the system, find the component in the list of available components and do one of the following:

- Double-click the component.
- Select the component, then click **Add**.
- Right-click the component and choose **Add New** <*component name*>.

If the component has any parameterized features, a wizard will appear allowing you to configure this instance of the component. Figure 2–4 shows an example of a configuration wizard for the SDRAM controller included with the Quartus II software.

Figure 2–4. SDRAM Controller Configuration Wizard


SDRAM Controller - sdram

Presets: single Micron MT48LC4M32B2-7 chip

Memory Profile | Timing

SDRAM Timing Parameters

CAS latency cycles	1 2 [✓ 3]	
Initialization refresh cycles	2	
Issue one refresh command every	15.625	us
Delay after powerup, before initialization	100	us
Duration of refresh command (t_rfc)	70	ns
Duration of precharge command (t_rp)	20	ns
ACTIVE to READ or WRITE delay (t_rcd)	20	ns
Access time (t_ac)	5.5	ns
Write recovery time (t_wr, No auto precharge)	14	ns

Like the overall SOPC Builder GUI, component configuration wizards use tabs to categorize GUI features, and have a message window that displays dynamic information about the current configuration of the component.

The parameters and information displayed in the configuration wizard depend on the component. Many wizards provide tool tips that give you information on how to specify each parameter.

☞ You can open component documentation from within SOPC Builder. Right-click the component in the list of available components, and choose one of the document items listed.

Specifying Connections, Base Address, Clock & IRQ

After you add a component, you must configure how it fits within the system.

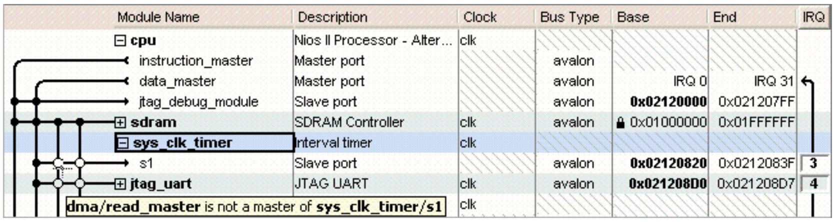
Connection Panel

In the connection panel you specify the master-slave connections between components.

☞ Each SOPC Builder component can have one or more master and slave ports. Each slave port must be connected to a master port.

Hovering the mouse over the connection panel displays the potential connection points between components, represented as dots connecting wires. A filled dot shows that a connection is made; an open dot shows a potential connection point that is not currently connected. Clicking a dot toggles the connection status. See Figure 2–5.

Figure 2–5. Connection Panel



Module Name	Description	Clock	Bus Type	Base	End	IRQ
cpu	Nios II Processor - Alter...	clk				
instruction_master	Master port		avalon			
data_master	Master port		avalon	IRQ 0	IRQ 31	
jtag_debug_module	Slave port		avalon	**0x02120000**	0x021207FF	
sdram	SDRAM Controller	clk	avalon	🔒 0x01000000	0x01FFFFFF	
sys_clk_timer	Interval timer	clk				
s1	Slave port		avalon	**0x02120820**	0x0212083F	3
jtag_uart	JTAG UART	clk	avalon	**0x021208D0**	0x021208D7	4
dma/read_master is not a master of **sys_clk_timer/s1**		clk				

You can connect any master port to any slave port, as long as the ports use the same type of hardware interface. If they use different interfaces, they must communicate through a bridge component, such as the AHB-to-Avalon bridge provided with SOPC Builder.

Table of Active Components

After your component is connected in the connection panel, you use the table of active components to specify the parameters listed in Table 2–2.

Table 2–2. Table of Active Components

Parameter	Description
Name	Name of the instance of the component. SOPC Builder assigns a default name when you add a new component. To change this name, right click the component name and select **Rename**.
Use	Enables/Disables the component in the current system. Turning off the **Use** setting is equivalent to removing the component from the system.
Clock	Clock source for the component. You must choose one of the clocks defined in the clock settings table.
Base *(1)*	Base address where the slave port will appear in the master port's address space. SOPC Builder can automatically choose new base address values to avoid conflicts between all components. You can automatically assign base addresses for all components by choosing **Auto-Assign Base Addresses** (System menu). You can lock the base address on a component so that SOPC Builder will not change it automatically. To lock the base address, right-click the component in the table of active components, and choose **Lock Base Address**.
IRQ *(1)*	Specifies the IRQ value the slave drives to the master, if the slave port can generate interrupts. If you specify **NC** for an IRQ value, SOPC Builder will not connect interrupt signals from slave to master. SOPC Builder can automatically choose new IRQ values to avoid conflicts between all components. To automatically assign IRQs for all components, choose **Auto-Assign IRQs** (System menu).

Note to Table 2–2:
(1) This setting applies to slave ports only.

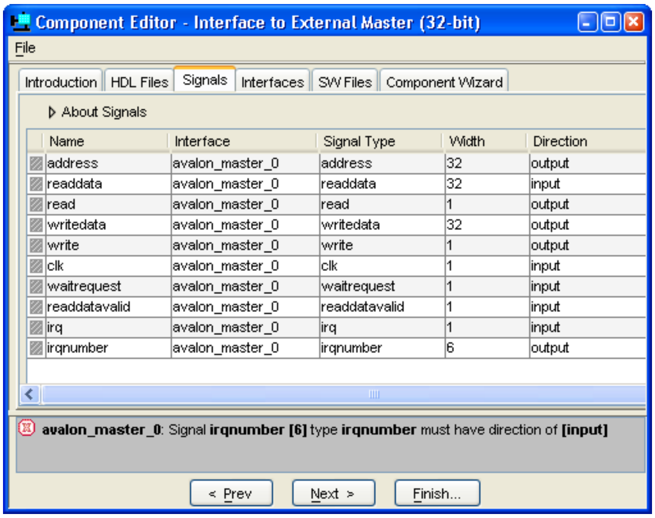
☞ SOPC Builder automatically chooses defaults for these parameters when you add each component, but you must verify that they are appropriate for your system.

Creating User-Defined Components

You can use the SOPC Builder component editor to create a component from user-defined logic. To open the component editor, choose **New Component** (File menu) or select **Create New Component** in the list of available components, and click **Add**.

Figure 2–6 shows an example of the component editor editing a 32-bit master component.

Figure 2–6. SOPC Builder Component Editor



Name	Interface	Signal Type	Width	Direction
address	avalon_master_0	address	32	output
readdata	avalon_master_0	readdata	32	input
read	avalon_master_0	read	1	output
writedata	avalon_master_0	writedata	32	output
write	avalon_master_0	write	1	output
clk	avalon_master_0	clk	1	input
waitrequest	avalon_master_0	waitrequest	1	input
readdatavalid	avalon_master_0	readdatavalid	1	input
irq	avalon_master_0	irq	1	input
irqnumber	avalon_master_0	irqnumber	6	output

avalon_master_0: Signal **irqnumber [6]** type **irqnumber** must have direction of **[input]**

After creating a user-defined component, the process to instantiate the component is the same as for any other component.

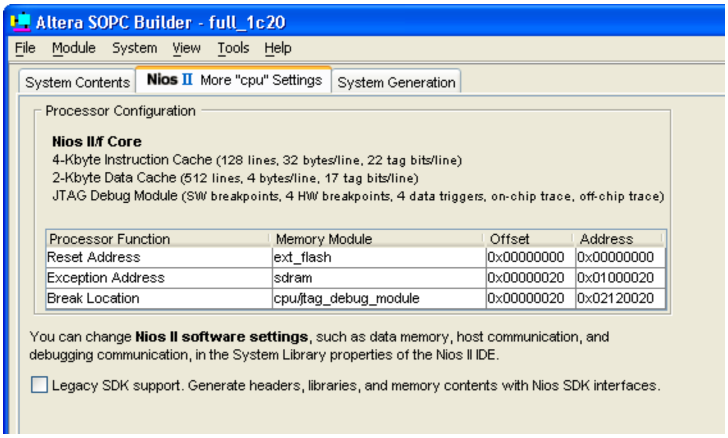
For instructions on developing a custom SOPC Builder component, see the *Developing SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For complete detail on the file structure of a component, see the *SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*.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

System Dependency Tabs

Certain components must be configured based on system-level factors external to the component. SOPC Builder provides system dependency tabs, which allow further configuration of a component beyond the component's configuration wizard. System dependency tabs are titled **More "**<*Name of component instance*>**" Settings**, and appear next to the **System Contents** tab.

Figure 2–7 shows an example of the system dependency tab for an instance of the Nios® II processor named `cpu`. In this example, the Nios II processor reset and exception addresses depend on memory components in the system but external to the processor, and therefore uses a system dependency tab.

Figure 2–7. System Dependency Tab for a Nios II Processor


Altera SOPC Builder - full_1c20
File Module System View Tools Help
System Contents
Nios II More "cpu" Settings
System Generation

Processor Configuration

Nios II/f Core
4-Kbyte Instruction Cache (128 lines, 32 bytes/line, 22 tag bits/line)
2-Kbyte Data Cache (512 lines, 4 bytes/line, 17 tag bits/line)
JTAG Debug Module (SW breakpoints, 4 HW breakpoints, 4 data triggers, on-chip trace, off-chip trace)

Processor Function	Memory Module	Offset	Address
Reset Address	ext_flash	0x00000000	0x00000000
Exception Address	sdram	0x00000020	0x01000020
Break Location	cpu/jtag_debug_module	0x00000020	0x02120020

You can change **Nios II software settings**, such as data memory, host communication, and debugging communication, in the System Library properties of the Nios II IDE.

☐ Legacy SDK support. Generate headers, libraries, and memory contents with Nios SDK interfaces.

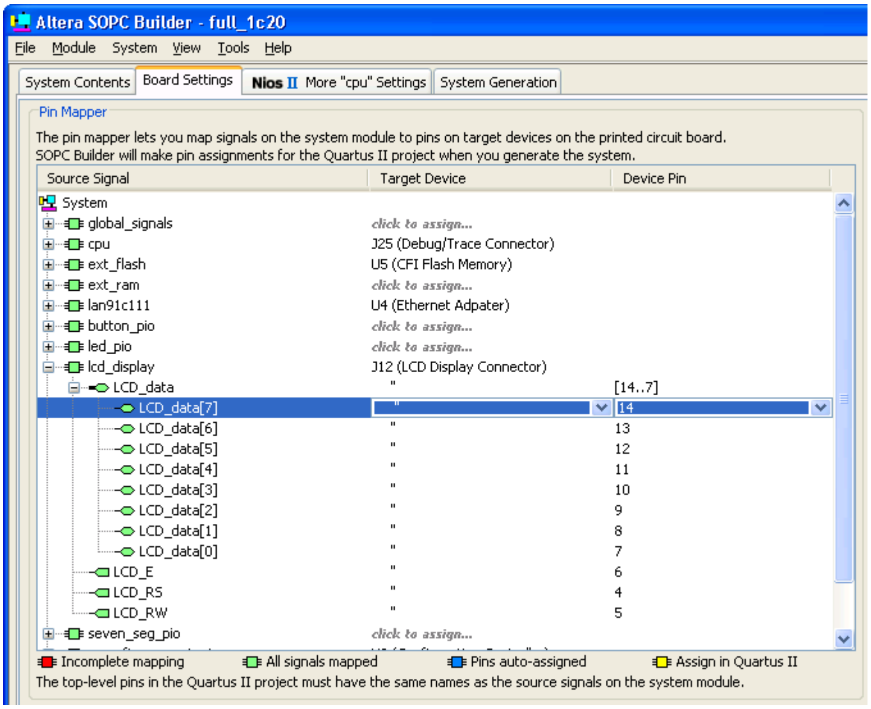
Board Settings Tab

The **Board Settings** tab provides board-specific settings for the target hardware platform. SOPC Builder displays a **Board Settings** tab only if you specify a target board on the **System Contents** tab.

The main feature of the **Board Settings** tab is the SOPC Builder pin mapper. The pin mapper simplifies the process of assigning FPGA pins. You use the pin mapper to map logical connections between system components and devices mounted on the PCB. Based on your pin mapper settings, SOPC Builder applies appropriate pin assignments to your Quartus II project.

In addition to accelerating the process of assigning FPGA pins, the pin mapper provides a level of assurance that you have accounted for the pin assignments required for a target board. Figure 2–8 on page 2–10 shows an example of the pin mapper for a target board.

Figure 2–8. Board Settings Tab


Altera SOPC Builder - full_1c20
File Module System View Tools Help
System Contents
Board Settings
Nios II More "cpu" Settings
System Generation
Pin Mapper

The pin mapper lets you map signals on the system module to pins on target devices on the printed circuit board.
SOPC Builder will make pin assignments for the Quartus II project when you generate the system.

Source Signal	Target Device	Device Pin
System		
global_signals	click to assign...	
cpu	J25 (Debug/Trace Connector)	
ext_flash	U5 (CFI Flash Memory)	
ext_ram	click to assign...	
lan91c111	U4 (Ethernet Adpater)	
button_pio	click to assign...	
led_pio	click to assign...	
lcd_display	J12 (LCD Display Connector)	
LCD_data	"	[14..7]
LCD_data[7]	"	14
LCD_data[6]	"	13
LCD_data[5]	"	12
LCD_data[4]	"	11
LCD_data[3]	"	10
LCD_data[2]	"	9
LCD_data[1]	"	8
LCD_data[0]	"	7
LCD_E	"	6
LCD_RS	"	4
LCD_RW	"	5
seven_seg_pio	click to assign...	

🟥 Incomplete mapping 🟩 All signals mapped 🟦 Pins auto-assigned 🟨 Assign in Quartus II

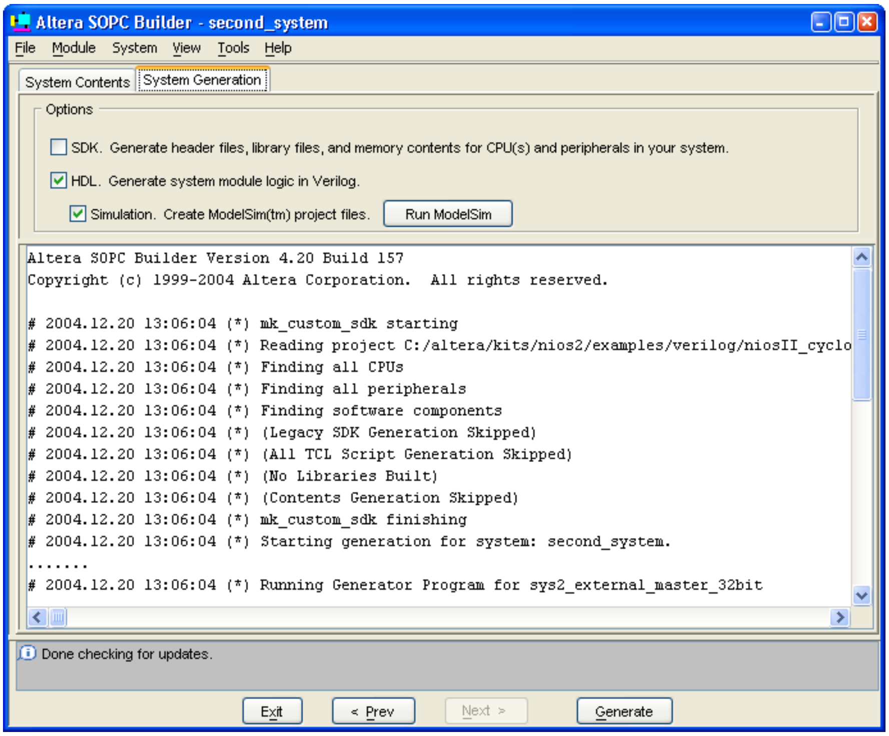
The top-level pins in the Quartus II project must have the same names as the source signals on the system module.

System Generation Tab

This section describes the settings available on the **System Generation** tab, and discusses the various outputs of SOPC Builder. System generation refers to SOPC Builder's process of generating output files that describe your system, based on the parameters you specified in the SOPC Builder GUI.

Figure 2–9 shows an example of the **System Generation** tab.

Figure 2–9. System Generation Page



SDK. Generate header files, library files, and memory contents for CPU(s) and peripherals in your system.

HDL. Generate system module logic in Verilog. [✓]

Simulation. Create ModelSim(tm) project files. [✓]

```
Altera SOPC Builder Version 4.20 Build 157
Copyright (c) 1999-2004 Altera Corporation.  All rights reserved.

# 2004.12.20 13:06:04 (*) mk_custom_sdk starting
# 2004.12.20 13:06:04 (*) Reading project C:/altera/kits/nios2/examples/verilog/niosII_cyclo
# 2004.12.20 13:06:04 (*) Finding all CPUs
# 2004.12.20 13:06:04 (*) Finding all peripherals
# 2004.12.20 13:06:04 (*) Finding software components
# 2004.12.20 13:06:04 (*) (Legacy SDK Generation Skipped)
# 2004.12.20 13:06:04 (*) (All TCL Script Generation Skipped)
# 2004.12.20 13:06:04 (*) (No Libraries Built)
# 2004.12.20 13:06:04 (*) (Contents Generation Skipped)
# 2004.12.20 13:06:04 (*) mk_custom_sdk finishing
# 2004.12.20 13:06:04 (*) Starting generation for system: second_system.
.......
# 2004.12.20 13:06:04 (*) Running Generator Program for sys2_external_master_32bit
```

Done checking for updates.

Some SOPC Builder components, such as the Nios II processor, may modify the available options on this page.

Depending on the options on the **System Generation** tab, SOPC Builder generates the following outputs:

- Hardware design files
- Simulation model, testbench, and ModelSim® project files
- Files for software development

System Generation Tab Options

This section describes each of the options available on the **System Generation** tab.

SDK Option

When the **SDK** option is turned on, during system generation SOPC Builder creates a custom software development kit (SDK) directory in the Quartus II project directory for each CPU in the system. The SDK contains a memory map and software files (drivers, libraries, and utilities) for any system components that provide software support files.

☞ The Nios II processor provides a different software development flow, and therefore does not use the **SDK** option.

SDK files are arranged into the following directories:

- **inc** – This directory contains header files. These files include the definition for the memory map, register declarations for included peripherals, and macros that can be used in creating embedded software applications.
- **lib** – This directory contains software library files. During system generation, processor components can include commands to have SOPC Builder compile the libraries automatically.
- **src** – This directory provides a location for application source code development. Example source code files associated with peripherals may also be copied into this directory during system generation.

Every time you generate or update the system hardware module, you must give the SDK files to the software engineers developing application code.

☞ If you edit the files generated by SOPC Builder, save them with a unique filename to prevent the file from being overwritten in a subsequent system generation.

HDL Option

When the **HDL** option is turned on, during system generation SOPC Builder creates HDL files that describe the system, and stores them in the Quartus II project directory. The HDL files contain the following:

- The top-level system module
- An instance of every component in the system
- The Avalon switch fabric tailored to connect all components in the system
- A simulation model and a simulation testbench, depending on the Simulation option. See "Simulation Option" on page 2–13 for more details.

SOPC Builder outputs the top-level system module file in either Verilog HDL or VHDL, depending on which language you specified when starting SOPC Builder. However, some SOPC Builder components may provide source files in only one language.

Simulation Option

When the **Simulation** option is turned on, during generation SOPC Builder creates a simulation model and a test bench for the system. Simulation-specific files are written to a simulation directory in the Quartus II project directory, separate from the synthesizable HDL files.

The testbench is tailored to the structure of system module. The testbench provides the following functionality:

- Instantiates the system module
- Drives clock and reset inputs with default behaviors
- Instantiates and connects the simulation models provided for any components external to the system module, such as memory models

Individual components may also provide simulation files, which SOPC Builder copies into the simulation directory during system generation.

SOPC Builder generates a ModelSim project directory that includes the following files:

- A ModelSim Project File (**.mpf**) for the current system
- Simulation data files for all memory components that have initialized contents
- A **setup_sim.do** file that contains setup information and aliases customized for simulating the system module
- A **wave_presets.do** file that automatically displays a default set of useful waveforms

You can run the ModelSim software directly from SOPC Builder by clicking **Run ModelSim**. This requires that the ModelSim software be specified in your search path in the **Setup** window (File menu).

☞ **Run ModelSim** might be disabled by a component in the system. For example, Nios II processor systems provide a different simulation flow, and therefore **Run ModelSim** is unavailable on the **System Generation** tab for Nios II processor systems.

The SOPC Builder output files for simulation are designed to work with the ModelSim simulator. You can use the SOPC Builder-generated simulation model and testbench with other HDL simulators. While the

ModelSim-specific files (**.tcl**, **.do**, **.mpf**, etc.) will not work directly with other simulators, you can inspect these files and use them as a basis for setting up similar capabilities.

Starting System Generation

After you have configured all system parameters and specified the desired generation options on this tab, clicking **Generate** starts the system generation process. **Generate** is available from all tabs in SOPC Builder. It is disabled whenever there are any errors displayed in the messages window.

The middle area of the **System Generation** tab is a progress display, showing a time-stamped list of progress messages that occur during system generation. SOPC Builder executes multiple tools and scripts to generate the system, and the status messages from each step are reported in the progress display.

If system generation completes successfully, SOPC Builder displays a final progress message:

```
# <timestamp> SUCCESS: SYSTEM GENERATION COMPLETED.
```

SOPC Builder saves a log file of the progress messages in the Quartus II project directory.

Other Tools

The Tools menu provides access to a dynamic list of downstream design tools, depending on the components in the system. For example, if the system contains a Nios II processor, Nios II processor-related tools are listed in the Tools menu.

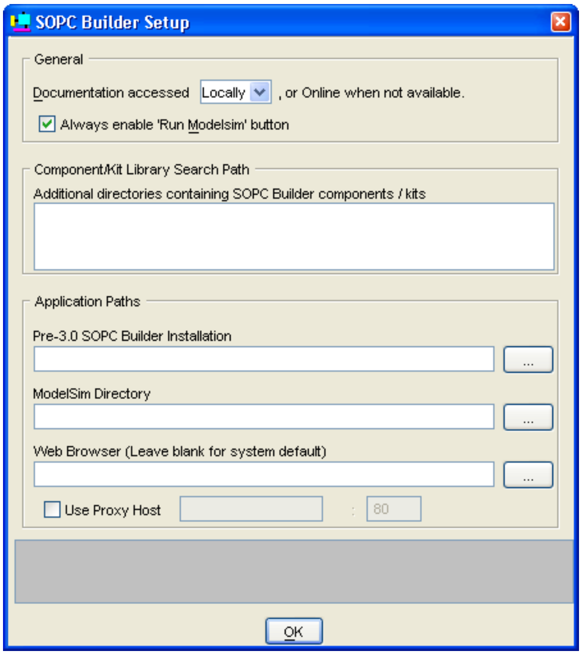
Preferences

The **SOPC Builder Setup** window (File menu) provides global options that affect SOPC Builder's operation, such as:

- Search path for SOPC Builder components
- Install path for ModelSim
- Location of documentation files
- Web browser settings (for accessing component documentation)

The **SOPC Builder Setup** window is shown in Figure 2–10.

Figure 2–10. SOPC Builder Setup Window



SOPC Builder Setup

General

Documentation accessed Locally, or Online when not available.

[✓] Always enable 'Run Modelsim' button

Component/Kit Library Search Path

Additional directories containing SOPC Builder components / kits

Application Paths

Pre-3.0 SOPC Builder Installation

ModelSim Directory

Web Browser (Leave blank for system default)

[ ] Use Proxy Host : 80

3. Avalon Switch Fabric

QII54003-6.0.0

Introduction

Avalon® switch fabric is a high-bandwidth interconnect structure that consumes minimal logic resources and provides greater flexibility than a typical shared system bus. This chapter describes the functions of Avalon switch fabric and the implementation of those functions.

High-Level Description

Avalon switch fabric is the glue that binds together components in a system based on the Avalon interface.

Avalon switch fabric is the collection of interconnect and logic resources that connects Avalon master and slave ports on components in a system. Avalon switch fabric encapsulates the connection details of a system. Avalon switch fabric guarantees that signals travel correctly between master and slave ports, as long as the ports adhere to the rules of the Avalon interface specification. As a result, system designers can think at a higher level and focus on the parts of a system that add value, rather than worry about the interconnect.

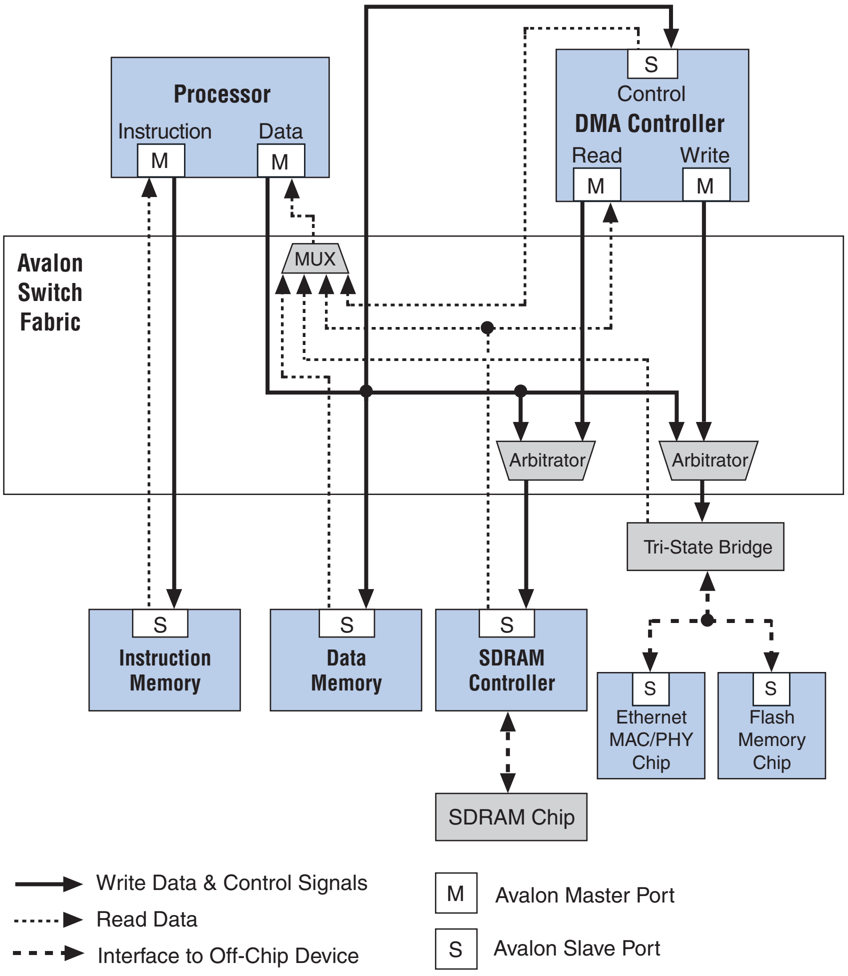
For details on the Avalon interface, refer to the *Avalon Interface Specification* available at **www.altera.com**. For details on how to use SOPC Builder to create Avalon switch fabric, refer to the *Tour of the SOPC Builder User Interface* chapter in volume 4 of the *Quartus II Handbook*.

Avalon switch fabric 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 peripherals
- Components of differing data widths
- Big-endian or little-endian components
- Components operating in different clock domains
- Components using multiple Avalon ports

Figure 3–1 shows a simplified diagram of the Avalon switch fabric in an example system with multiple masters.

Figure 3–1. Avalon Switch Fabric Block Diagram – Example System



Some components in Figure 3–1 use multiple Avalon ports. Because an Avalon component can have multiple Avalon ports, you can use Avalon switch fabric to create *super interfaces* that provide more functionality than a single Avalon port. For example, an Avalon slave port can have only one interrupt-request (IRQ) signal. However, by using three Avalon slave ports together, you can create a component that generates three separate IRQs. In this case, SOPC Builder generates the Avalon switch fabric to connect all ports.

Generating Avalon switch fabric is SOPC Builder's primary purpose. Because SOPC Builder generates Avalon switch fabric automatically, most users do not interact directly with it or the HDL that describes it. You do not need to know anything about the internal workings of Avalon switch fabric to take advantage of the services it provides. On the other hand, 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 multi-master systems minimize the impact of arbitration on the system throughput.

Fundamentals of Avalon Switch Fabric Implementation

Avalon switch fabric uses active logic to implement a switched interconnect structure that provides a dedicated path between master and slave ports. Avalon switch fabric consists of synchronous logic and routing resources inside an FPGA.

At each port interface, Avalon switch fabric manages Avalon transfers, responding to signals from the connected component. The signals that appear on the master port and corresponding slave port during a transfer can be very different, depending on how the Avalon switch fabric transports signals between the master-slave pair. In the path between master and slave ports, the Avalon switch fabric can introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by those ports.

Functions of Avalon Switch Fabric

Avalon switch fabric logic provides the following functions:

- Address Decoding (page 3–4)
- Data-Path Multiplexing (page 3–5)
- Wait-State Insertion (page 3–6)
- Pipelining for High Performance (page 3–7)
- Pipeline Management (page 3–8)
- Endian Conversion (page 3–9)
- Native Address Alignment & Dynamic Bus Sizing (page 3–10)
- Arbitration for Multi-Master Systems (page 3–13)
- Burst Management (page 3–20)
- Clock Domain Crossing (page 3–21)
- Interrupt Controller (page 3–25)
- Reset Distribution (page 3–27)

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 the SOPC Builder GUI. The remaining sections of this chapter describe how SOPC Builder implements each function.

Address Decoding

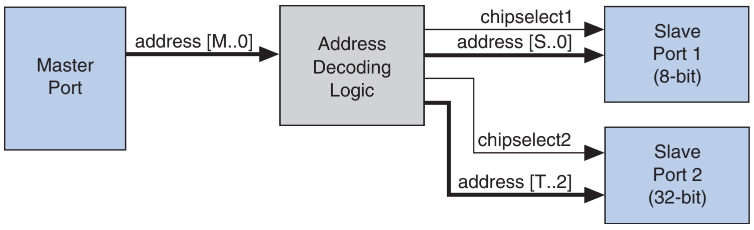
Address decoding logic in the Avalon switch fabric distributes an appropriate address and produces a chipselect signal for each slave port. Address decoding logic simplifies component design in the following ways:

- The Avalon switch 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 always properly aligned for the data width of 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 3–2 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 3–2, the address decoding logic handles the difference between the master address width (M) and the individual slave address widths (S & T). It also maps only the necessary master address bits to access words in each slave port's address space.

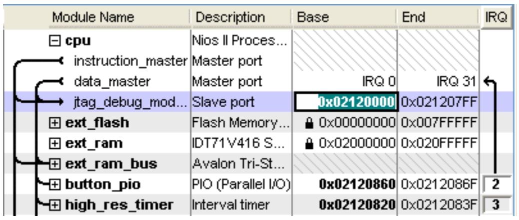
Figure 3–2. Block Diagram of Address Decoding Logic



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

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

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



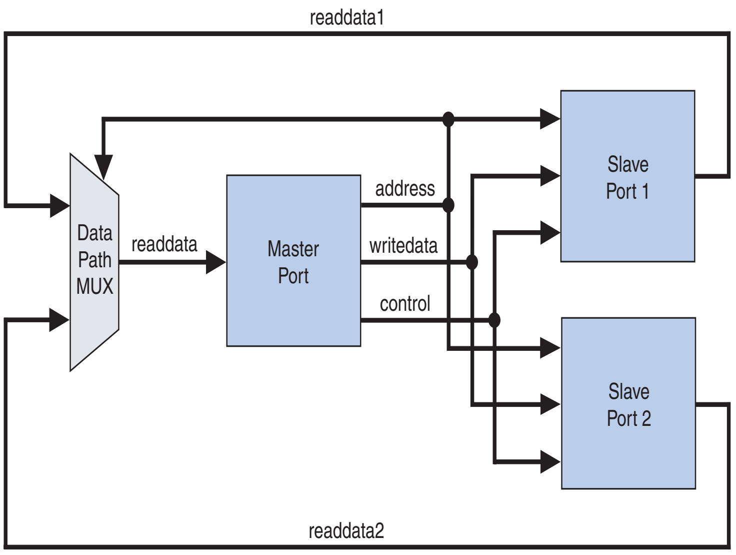
Module Name	Description	Base	End	IRQ
⊟ **cpu**	Nios II Proces...			
instruction_master	Master port			
data_master	Master port	IRQ 0	IRQ 31	
jtag_debug_mod...	Slave port	0x02120000	0x021207FF	
⊞ **ext_flash**	Flash Memory...	0x00000000	0x007FFFFF	
⊞ **ext_ram**	IDT71V416 S...	0x02000000	0x020FFFFF	
⊞ **ext_ram_bus**	Avalon Tri-St...			
⊞ **button_pio**	PIO (Parallel I/O)	**0x02120860**	0x0212086F	2
⊞ **high_res_timer**	Interval timer	**0x02120820**	0x0212083F	3

Data-Path Multiplexing

Data-path multiplexing logic in the Avalon switch fabric aggregates read-data signals from multiple slave ports during a read transfer, and presents the signals from only the selected slave back to the master port.

Figure 3–4 shows a block diagram of the data-path multiplexing logic for one master and two slave ports. SOPC Builder generates separate data-path multiplexing logic for every master port in the system.

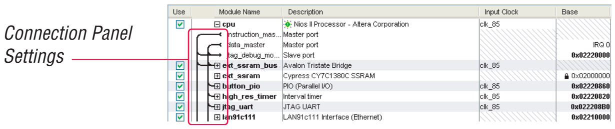
Figure 3–4. Block Diagram of Data-Path Multiplexing Logic



Data-path multiplexing is not necessary in the write-data direction for write transfers. The `writedata` signals are distributed equally to all slave ports, and each slave port ignores `writedata` except for when the address-decoding logic selects that port.

In the SOPC Builder GUI, the generation of data-path multiplexing logic is specified using the connections panel on the **System Contents** page, as shown in Figure 3–5.

Figure 3–5. Connection Panel Settings in SOPC Builder Control Data-Path Multiplexing



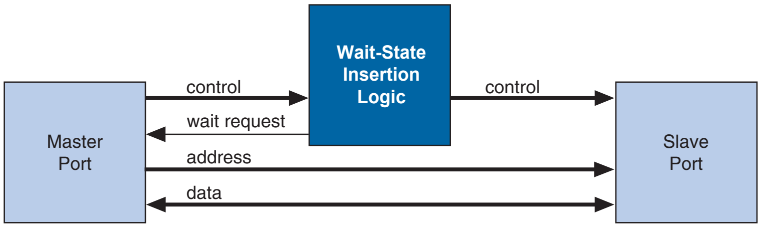
Use	Module Name	Description	Input Clock	Base
✓	cpu	Nios II Processor - Altera Corporation	clk_85	
	instruction_mas...	Master port		
	data_master	Master port		IRQ 0
	jtag_debug_mo...	Slave port		0x02220000
✓	ext_ssram_bus	Avalon Tristate Bridge	clk_85	
✓	ext_ssram	Cypress CY7C1380C SSRAM		0x02000000
✓	button_pio	PIO (Parallel I/O)	clk_85	0x02220860
✓	high_res_timer	Interval timer	clk_85	0x02220820
✓	jtag_uart	JTAG UART	clk_85	0x022208B0
✓	lan91c111	LAN91c111 Interface (Ethernet)		0x02210000

Wait-State Insertion

Wait states extend the duration of a transfer by one or more cycles for the benefit of components with special synchronization needs. Wait-state insertion logic accommodates the timing needs of each slave port, and coordinates the master port to wait until the slave can proceed. Avalon switch fabric inserts wait states into a transfer when the target slave port cannot respond in a single clock cycle. Avalon switch 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 3–6 shows a block diagram of the wait-state insertion logic between one master and one slave.

Figure 3–6. Block Diagram of Wait-State Insertion Logic



Avalon switch 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 multi-master 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.

Pipelining for High Performance

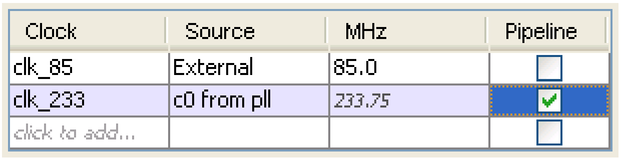
SOPC Builder can pipeline the Avalon switch fabric by inserting stages of registers between master-slave pairs. Adding pipeline registers can increase the f_{MAX} performance of the system and ensure that the critical timing path does not occur inside the Avalon switch fabric.

The pipeline registers introduce one or more clock cycles of latency between master-slave pairs, which creates a trade-off between transfer latency and f_{MAX} performance. The pipeline registers can also increase logic utilization considerably, depending on the complexity of the system. Components that support pipelined Avalon transfers minimize the effects of the pipeline latency. For details on how pipelining for high performance affects pipelined Avalon ports, see section "Pipeline Management" on page 3–8.

Pipeline registers are most likely to improve performance for the case of many master ports sharing a common slave port. For N masters accessing a slave port, the increased latency is on the order of $(\log_2 N + 1)$.

You specify whether or not to add pipelining for high performance with the clock settings table on the **System Contents** tab in SOPC Builder, shown in Figure 3–7. You can pipeline each clock domain separately by turning on its **Pipeline** check box.

Figure 3–7. Turning On Pipelining for High Performance



Clock	Source	MHz	Pipeline
clk_85	External	85.0	☐
clk_233	c0 from pll	233.75	✓
click to add...			☐

Pipeline Management

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

SOPC Builder generates Avalon switch 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, Avalon switch fabric often contains logic to reconcile the differences. Many cases are possible, as shown in Table 3–1.

Table 3–1. Various Cases of Pipeline Latency in a Master-Slave Pair (Part 1 of 2)

Master Port	Slave Port	Pipeline Management Logic Structure
No Pipeline	No Pipeline	The Avalon switch fabric does not instantiate logic to handle pipeline latency.
No Pipeline	Pipelined with Fixed or Variable Latency	The Avalon switch 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 Avalon switch fabric carries out the transfer as if neither port were pipelined, forcing the master port to wait until the slave port returns data.

Table 3–1. Various Cases of Pipeline Latency in a Master-Slave Pair (Part 2 of 2)

Master Port	Slave Port	Pipeline Management Logic Structure
Pipelined	Pipelined with Fixed Latency	The Avalon switch 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 data 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 GUI settings that directly control the pipeline management logic in the Avalon switch fabric.

Endian Conversion

Starting with version 5.1 of the Quartus II software, SOPC Builder supports big endian master ports. Prior to version 5.1, SOPC Builder treated all components as little endian. With version 5.1 and later, an Avalon-based system can contain both big and little endian components.

The endianness of an Avalon port depends on the component design. Endianness affects the order a master port expects individual bytes to be arranged within a larger word. If all master ports in the system use the same endianness, then all master ports' perception of byte addresses is consistent within the system. In this case there is no further endian-related design consideration required.

Avalon switch fabric provides endian-conversion functionality to allow master ports of differing endianness to share memory. The Avalon endian-conversion logic hides the endian difference of master ports when the following conditions are met:

1. The master ports access a common memory slave port.
2. The data width of the master ports are equal.
3. The master ports read and write the memory using only native-width units of data. For example, a 32-bit master port can read and write only 32-bit units of data.
4. The master ports use a common interpretation of the data type.

As an example, consider a three-chip system comprised of a discrete 32-bit CPU chip, an FPGA containing 32-bit coprocessor logic (an SOPC Builder system), and a shared DDR SDRAM chip. Furthermore, suppose that the CPU is big endian, while the FPGA coprocessor system is little endian. In this case, the CPU and the coprocessor can share data in the SDRAM seamlessly without manually accounting for the endianness of data.

☞ The Avalon switch fabric does not guarantee proper byte arrangement for big-endian master ports when accessing peripheral registers via an Avalon slave port.

Native Address Alignment & Dynamic Bus Sizing

SOPC Builder generates Avalon switch 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 it may 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 3–2 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 0x0000000 in the master port. In Table 3–2, `OFFSET` refers to the offset into the 16-bit slave port address space.

Table 3–2. 32-Bit Master View of 16-Bit Slave Data

32-bit Master Address	Data with Native Alignment	Data with Dynamic Bus Sizing
`0x00000000` (word 0)	`0x0000:OFFSET[0]`	`0xOFFSET[1]:OFFSET[0]`
`0x00000004` (word 1)	`0x0000:OFFSET[1]`	`0xOFFSET[3]:OFFSET[2]`
`0x00000008` (word 2)	`0x0000:OFFSET[2]`	`0xOFFSET[5]:OFFSET[4]`
`0x0000000C` (word 3)	`0x0000:OFFSET[3]`	`0xOFFSET[7]:OFFSET[6]`
...		
(word *N*)	`0x0000:OFFSET[N]`	`0xOFFSET[2N+1]:OFFSET[2N]`

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 GUI settings that directly control the address alignment in the Avalon switch fabric.

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, Avalon switch fabric maps all slave data bits to the lower bits of the master data, and fills any remaining upper bits with zero. Avalon switch fabric performs simple wire-mapping in the data path, 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.

Dynamic Bus Sizing

Slave ports that access memory devices generally use dynamic bus sizing. Dynamic bus sizing hides the details of interfacing a narrow memory 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 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 seamlessly, regardless of the data width.

In the case of dynamic bus sizing, the Avalon switch 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 *N* slave transfers. The master port pays a performance penalty, because it must wait while multiple slave-side transfers complete.

In the case of a read transfer, the Avalon switch fabric merges slave data from multiple read transfers before presenting them to the master port. A read transfer from a wide master port always causes multiple slave-read transfers to sequential addresses in the slave's address space. For example, even if a 32-bit master port needs only one byte from a dynamically-aligned 8-bit memory, a master read transfer generates four slave transfers, and the master port waits until all four transfers complete.

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

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. There is no performance penalty when accessing a wider slave port using dynamic bus sizing.

Table 3–3 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.

Table 3–3. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing

32-bit Address	Data
`0x00000000` (word 0)	`OFFSET[0]` $_{31..0}$
`0x00000004` (word 1)	`OFFSET[0]` $_{63..32}$
`0x00000008` (word 2)	`OFFSET[1]` $_{31..0}$
`0x0000000C` (word 3)	`OFFSET[1]` $_{63..32}$

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

Arbitration for Multi-Master Systems

Avalon switch fabric supports systems with multiple master components. In a system with multiple master ports, such as the system pictured in Figure 3–1 on page 3–2, the Avalon switch 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 multi-master architecture used by Avalon switch 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 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 the SOPC Builder GUI. 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 switch fabric. Each Avalon master port connects to the Avalon switch fabric like it is the only master port in the system. As a result, you can reuse a component in single-master and multi-master systems without requiring design changes to the component.

This section discusses the architecture of the Avalon switch fabric generated by SOPC Builder for multi-master systems.

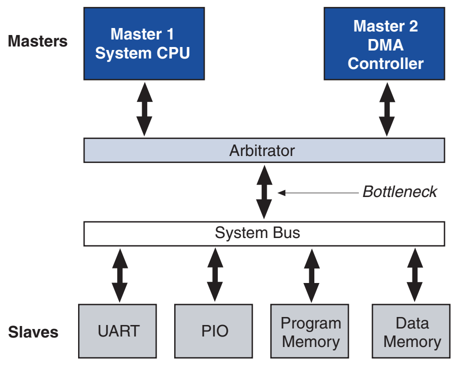
Traditional Shared Bus Architectures

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 arbitrator controls the bus (that is, the path between bus masters and bus slaves), so that multiple bus masters do not simultaneously drive the bus and cause electrical contention. Each bus master requests control of the bus from the arbitrator, and the arbitrator 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 arbitrator 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 arbitrator always grants control to the master with the highest priority—is used in many existing bus architectures.

Figure 3–8 illustrates the bus architecture for a traditional processor system. Access to the shared system bus becomes the bottleneck for throughput and utilization performance. Only one master has access to the bus at a time, which means that other masters are forced to wait (diminishing throughput), and only one slave can transfer data at a time (diminishing utilization).

Figure 3–8. Bus Architecture in a Traditional Microprocessor System



Slave-Side Arbitration

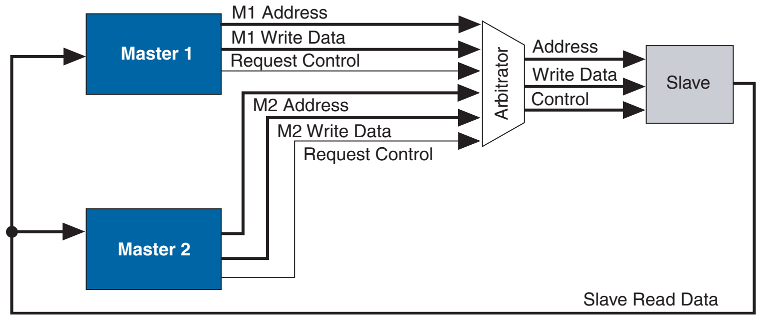
The multi-master architecture used by Avalon switch fabric eliminates the bottleneck for access to a shared bus, because the system does not have shared bus lines. Avalon master-slave pairs are connected by dedicated paths. A master port never waits to access a slave port, unless a different master port 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 multi-master Avalon 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 3–1 on page 3–2 demonstrates a system with two master ports (a CPU and a DMA controller) sharing a slave port (an SDRAM controller). Arbitration is performed on the SDRAM slave port; the arbitrator dictates which master port gains access to the slave port if both master ports initiate a transfer with the slave port at the same time.

Figure 3–9 focuses on the two master ports and the shared slave port, and shows additional detail of the data, address, and control paths. The arbitrator logic multiplexes all address, data, and control signals from a master port to a shared slave port.

Figure 3–9. Detailed View of Multi-Master Connections



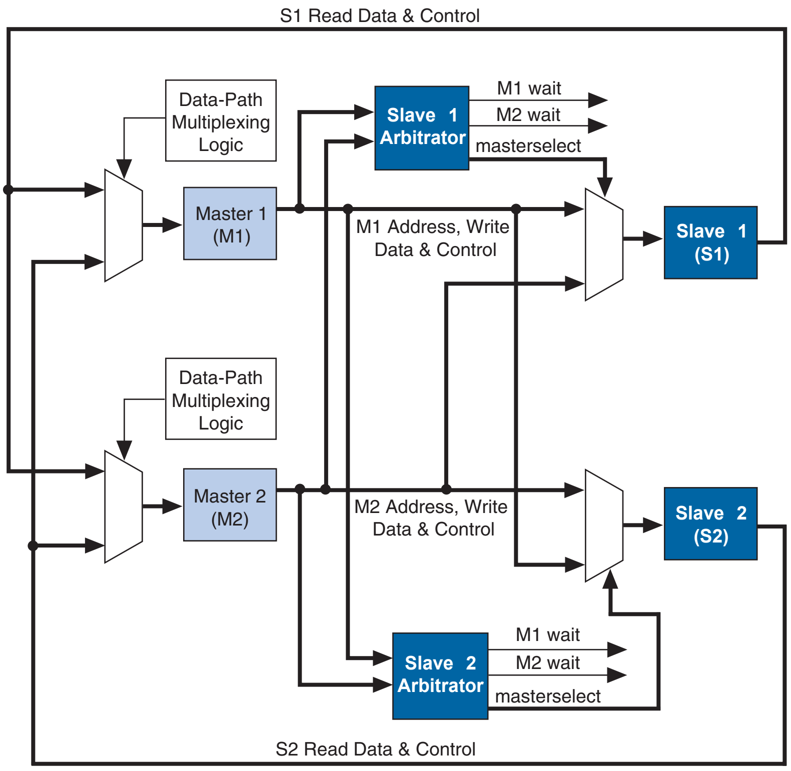
Arbitrator Details

SOPC Builder generates an arbitrator for every slave port connected to multiple master ports, based on arbitration parameters specified in the SOPC Builder GUI. The arbitrator logic performs the following functions for its associated slave port:

- Evaluates the address and control signals from each master port at every clock cycle when a new transfer can begin, and determines which master port, if any, is requesting access to the slave.
- Chooses which master port gains access to the slave next.
- Grants access to the chosen master port (that is, allows it to proceed with the transfer), and forces all other requesting master ports to wait.
- Uses multiplexers to connect address, control, and data paths between the multiple master ports and the slave port. The arbitrator logic guarantees that an appropriate master port (if any) is connected to the slave port.

Figure 3–10 shows the arbitrator logic in an example multi-master system with two master ports, each connected to two slave ports.

Figure 3–10. Block Diagram of Arbitrator Logic



Arbitration Rules

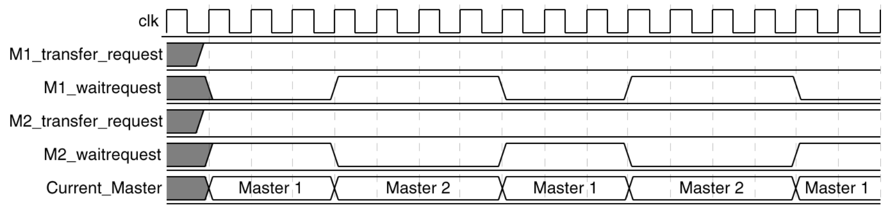
This section describes the rules by which the arbitrator grants access to master ports when they contend.

Fairness-Based Shares

Avalon arbitrator 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.

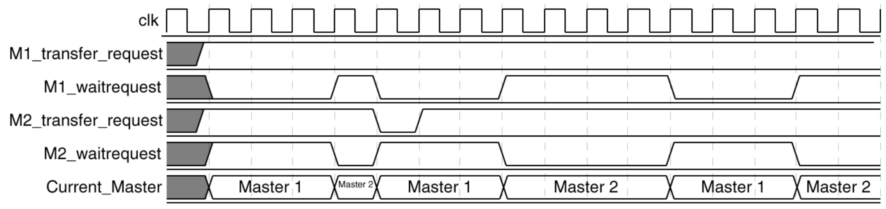
For example, assume that two master ports continuously attempt to perform back-to-back transfers to a slave port. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbitrator grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. Figure 3–11 demonstrates this case, showing each master port's transfer request output, wait request input (which is driven by the arbitrator logic), and the current master with control of the slave.

Figure 3–11. 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 arbitrator grants access to another requesting master. See Figure 3–12. After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbitrator grants access back to Master 1, which gets a replenished supply of shares.

Figure 3–12. 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 arbitrator grants shares in round-robin order. At every slave transfer, only requesting master ports are included in the round-robin arbitration.

Burst Transfers

Avalon 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, arbitrator logic does not allow any other master port to access the slave port until the burst completes. For further information, see "Burst Management" on page 3–20.

Minimum Share Value

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

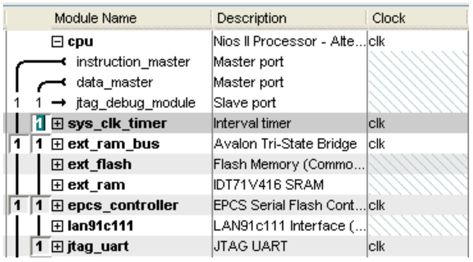
By declaring a minimum share value of *N*, a slave port declares that it is more efficient at handling continuous sequential transfers of length *N*. Accessing the slave port in sequences less than *N* incurs performance penalties that might prevent the slave port from achieving higher performance. By 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. See "Burst Management" on page 3–20 for information.

Setting Arbitration Parameters in the SOPC Builder GUI

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of the SOPC Builder GUI, as shown in Figure 3–13.

Figure 3–13. 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	
1	1	jtag_debug_module	Slave port	
	1	sys_clk_timer	Interval timer	clk
1	1	ext_ram_bus	Avalon Tri-State Bridge	clk
		ext_flash	Flash Memory (Commo...	
		ext_ram	IDT71V416 SRAM	
1	1	epcs_controller	EPCS Serial Flash Cont...	clk
		lan91c111	LAN91c111 Interface (...	
	1	jtag_uart	JTAG UART	clk

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

Burst Management

Avalon switch 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 Avalon arbitrator 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 Avalon switch fabric upholds the Avalon protocol with each port independently, and therefore each Avalon port need only be aware of its own clock domain. The Avalon switch fabric logic propagates transfers across clock domain boundaries transparently to the user.

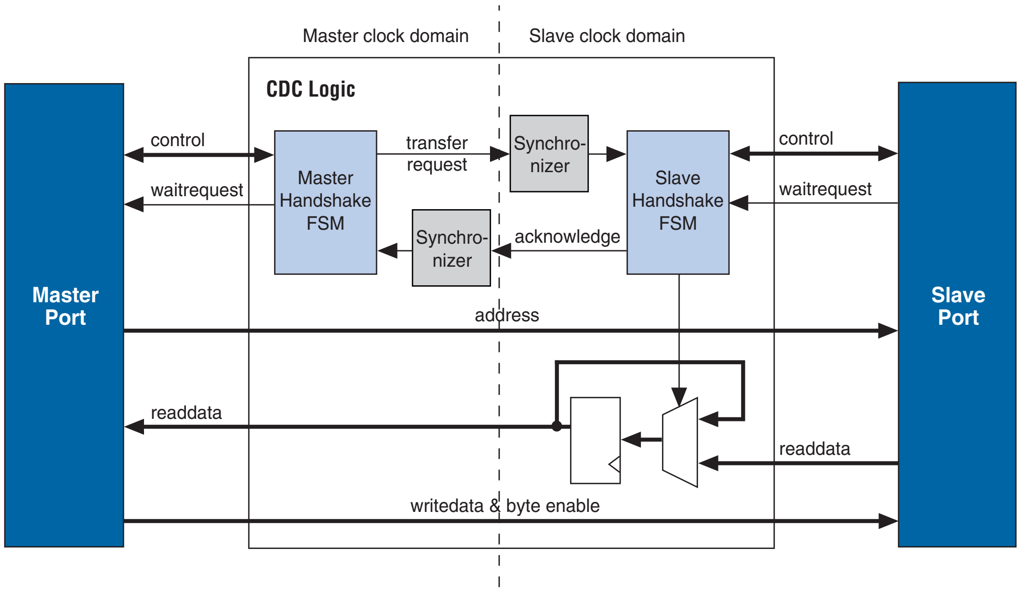
The CDC logic in Avalon switch fabric provides the following benefits that simplify system design efforts:

- Allows component interfaces to operate at a different clock frequency than system logic.
- Eliminates the need to design CDC hardware manually.
- Each Avalon port operates in only one clock domain, which reduces design complexity of components.
- Enables master ports to access any slave port without awareness of 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, which 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 3–14 shows a block diagram of the clock domain crossing logic between one master and one slave port.

Figure 3–14. Block Diagram of Clock Domain-Crossing Logic



The Synchronizer blocks in Figure 3–14 use multiple stages of flip-flops to eliminate the propagation of metastable events on the control signals that enter the hand-shake 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 takes extra clock cycles. Similar to other transfer delay cases (for example, arbitration delay and/or wait states on the slave side), the Avalon switch 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

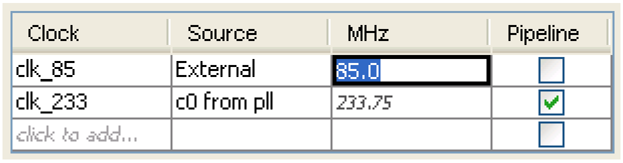
CDC logic extends the duration of master transfers across clock domain boundaries. In the worst case, each transfer is extended by five master clock cycles and five slave clock cycles. The components of this delay are the following:

- 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

Implementing Multiple Clock Domains in the SOPC Builder GUI

You specify the clock domains used by your system on the **System Contents** tab of the SOPC Builder GUI. You define the input clocks to the system using the clock settings table, shown in Figure 3–15. 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.

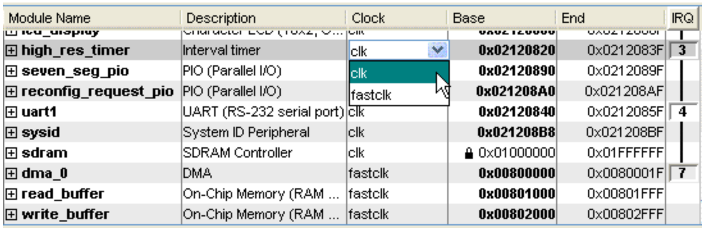
Figure 3–15. Clock Settings on the System Contents Tab



Clock	Source	MHz	Pipeline
clk_85	External	85.0	☐
clk_233	c0 from pll	233.75	✓
click to add...			☐

After you define the system clocks, you specify which clock drives which components using the table of active components, as shown in Figure 3–16.

Figure 3–16. 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	

For further details, refer to the *Building Systems with Multiple Clock Domains* chapter in volume 4 of the *Quartus II Handbook.*

Interrupt Controller

In systems with one or more slave ports that generate IRQs, the Avalon switch 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.

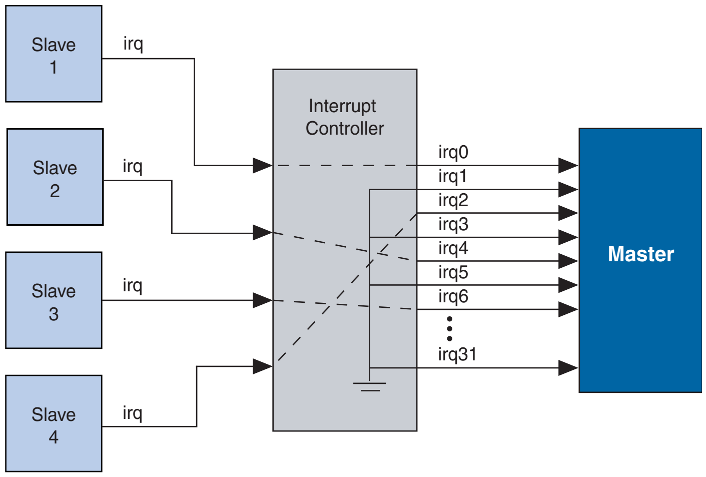
Each slave port optionally produces an IRQ output signal. There are two master signals related to interrupts: `irq` and `irqnumber`. SOPC Builder generates the interrupt controller in one of two configurations, software priority or hardware priority, depending on the interrupt signals present on the master port.

Software Priority

In the software priority configuration, the Avalon switch 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 3–17 shows an example of the interrupt controller mapping the IRQs on four slave ports to `irq[31..0]` on a master port.

Figure 3–17. IRQ Mapping Using Software Priority

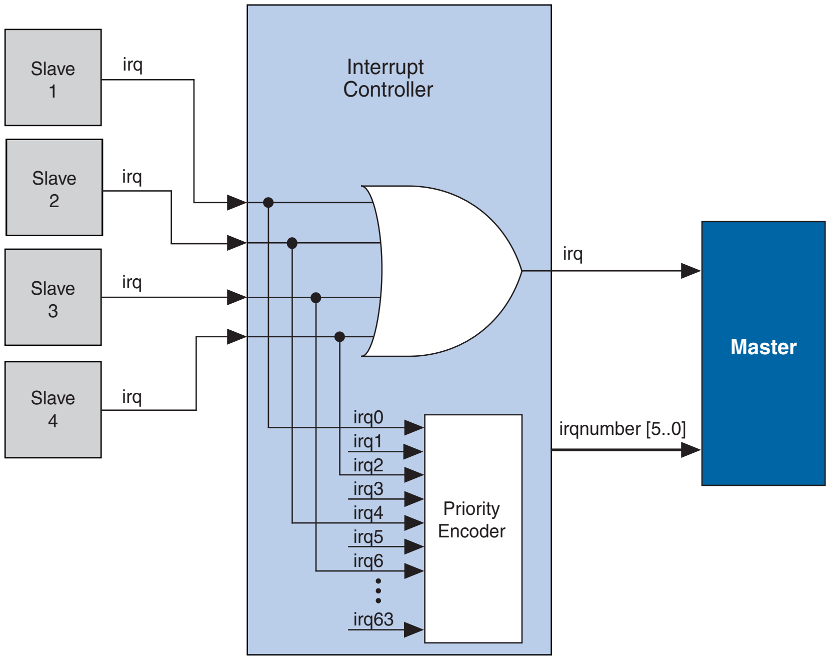


Hardware Priority

In the hardware priority configuration, in the event that multiple slaves assert their IRQs simultaneously, the Avalon switch fabric (i.e. 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 3–18.

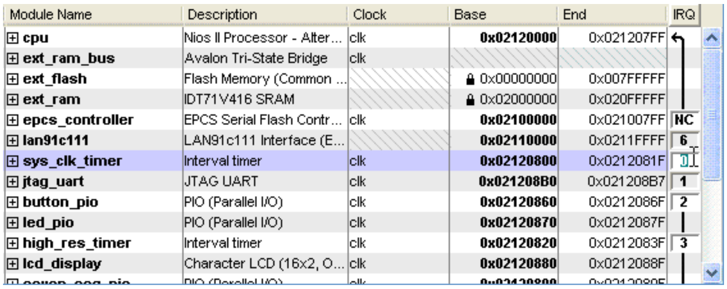
Figure 3–18. IRQ Mapping Using Hardware Priority



Assigning IRQs in the SOPC Builder GUI

You specify IRQ settings on the **System Contents** tab of the SOPC Builder GUI. 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 3–19 shows the IRQ settings for multiple slave IRQs that drive the master component named **cpu**.

Figure 3–19. Assigning IRQs in the SOPC Builder GUI



Module Name	Description	Clock	Base	End	IRQ
⊞ **cpu**	Nios II Processor - Alter...	clk	**0x02120000**	0x021207FF	
⊞ **ext_ram_bus**	Avalon Tri-State Bridge	clk			
⊞ **ext_flash**	Flash Memory (Common ...		0x00000000	0x007FFFFF	
⊞ **ext_ram**	IDT71V416 SRAM		0x02000000	0x020FFFFF	
⊞ **epcs_controller**	EPCS Serial Flash Contr...	clk	**0x02100000**	0x021007FF	**NC**
⊞ **lan91c111**	LAN91c111 Interface (E...		**0x02110000**	0x0211FFFF	**6**
⊞ **sys_clk_timer**	Interval timer	clk	**0x02120800**	0x0212081F	0
⊞ **jtag_uart**	JTAG UART	clk	**0x021208B0**	0x021208B7	**1**
⊞ **button_pio**	PIO (Parallel I/O)	clk	**0x02120860**	0x0212086F	**2**
⊞ **led_pio**	PIO (Parallel I/O)	clk	**0x02120870**	0x0212087F	
⊞ **high_res_timer**	Interval timer	clk	**0x02120820**	0x0212083F	**3**
⊞ **lcd_display**	Character LCD (16x2, O...	clk	**0x02120880**	0x0212088F	

Reset Distribution

The Avalon switch fabric generates and distributes a system-wide reset pulse to all logic in the system module. The switch fabric distributes the reset signal conditioned for each clock domain. The duration of the reset signal is at least one clock period.

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

- The global reset input to the system module is asserted.
- A slave port asserts its `resetrequest` signal.
- The FPGA is reconfigured.

All components must enter a well-defined reset state whenever the Avalon switch fabric asserts the system-wide reset. The timing of the reset signal is asynchronous to the operation of transfers.

4. SOPC Builder Components

QII54004-6.0.0

Introduction

This chapter describes in detail what an SOPC Builder component is. SOPC Builder components are individual design blocks that SOPC Builder uses to integrate a larger system module. Each component consists of a structured set of files within a directory.

The files in a component directory serve the following purposes:

- Defines the hardware interface to the component, such as the names and types of I/O signals.
- Declares any parameters that specify the structure of the component logic and the component interface.
- Describes a configuration wizard GUI for configuring the component in SOPC Builder.
- Provides scripts and other information SOPC Builder needs to generate the component HDL and integrate the component into the system module.
- Contains component-related information, such as software drivers, necessary for development steps downstream from SOPC Builder.

For details on creating custom components, see the *Developing SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook*. For details on the SOPC Builder component editor, see the *Component Editor* chapter in Volume 4 of the *Quartus II Handbook*.

Sources of Components

There are several sources for components, including the following:

- The Quartus® II software, which includes SOPC Builder, installs a number of components.
- Altera® development kits, such as the Nios® II Development Kit, provide SOPC Builder components as features.
- Third-party developers provide SOPC Builder Ready components, including component directories and documentation on how to use the component.
- You can package your own HDL files into a new, custom component, using the SOPC Builder component editor.

While it is possible to write component files manually, Altera strongly recommends you use the SOPC Builder component editor to create custom components, for reasons of consistency and forward compatibility.

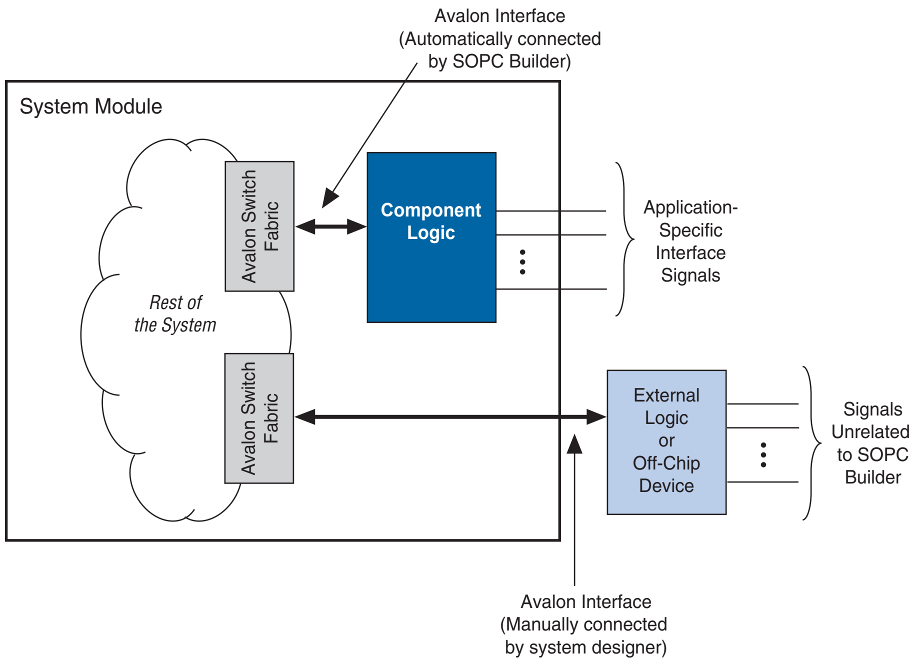
Location of the Component Hardware

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 component.

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


Avalon Interface
(Automatically connected by SOPC Builder)
System Module
Avalon Switch Fabric
Component Logic
Application-Specific Interface Signals
Rest of the System
Avalon Switch Fabric
External Logic or Off-Chip Device
Signals Unrelated to SOPC Builder
Avalon Interface
(Manually connected by system designer)

Components That Include Logic Inside the System Module

In this case, the component files provide a full description of the component hardware. During system generation, SOPC Builder instantiates the component logic inside the system module and automatically wires the component to the rest of the system. Internal to the system module, the component connects to the rest of the system through its Avalon® interface. The component can also have non-Avalon signals that SOPC Builder exposes on the top-level system module.

Components That Interface to Logic Outside the System Module

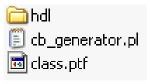
In this case, the component files describe only the interface to logic external to the system module. During system generation, SOPC Builder does not instantiate any logic for this component. Instead, SOPC Builder exposes an Avalon interface for this component on the top-level system module. You must manually wire the interface to external logic, such as a separate HDL module or an off-chip device.

As shown in Figure 4–1, in this case there is no physical embodiment of the SOPC Builder component. Components that interface to external logic describe only the shape of the Avalon interface; they do not include logic inside the system module.

Structure & Contents of a Component Directory

This section describes the files that exist in a component directory. Figure 4–2 shows a typical component directory created by the component editor.

Figure 4–2. Typical Component Directory



At a minimum, a component consists of a directory containing a file named **class.ptf**. Usually there are other files as well. The name of the enclosing directory is ignored. The **class.ptf** file defines everything that SOPC Builder needs to know about the name and location of component files.

As shown in Figure 4–2, the component directory can contain the following items, and other files:

- **class.ptf** file
- **cb_generator.pl** script
- **hdl** directory

class.ptf File

SOPC Builder reads this file to find everything it needs to know about the component. The **class.ptf** file defines the following aspects of a component:

1. The component identity – The component **class.ptf** file defines the following properties that identify a component:

- *Name* – The user-visible name of the component.
- *Class* – The unique identifier that SOPC Builder uses to differentiate components. The class name uses only filename-friendly characters. When you instantiate a component in SOPC Builder, the instantiation name defaults to the class name.
- *Version* – Identifies the version of a component. During system generation, SOPC Builder records the version number of each component in the system module.
- *Group* – Determines in which group the component appears in the list of available components in the SOPC Builder GUI.

2. How the component hardware connects into a system module, including:

   - *Interface to the component* – The **class.ptf** file describes the black box structure of the component, defining the name, type, and direction of each interface signal.
   - *Method to generate a hardware instance of the component* – If the component includes logic inside the system module, the **class.ptf** file specifies a mechanism for SOPC Builder to generate the component hardware. The **class.ptf** file typically specifies a separate executable file to generate the component instance.

3. How the component presents its configurable options to the user – The **class.ptf** file declares the user-configurable parameters, and specifies a GUI for configuring those parameters.

cb_generator.pl File

This file is a Perl script used by SOPC Builder during system generation. Based on parameters you specify in the SOPC Builder GUI, the script generates one or more instances of this component to be instantiated in the top-level system module. The name of the file is not significant. The file name is defined in the **class.ptf** file.

For components generated by the SOPC Builder component editor, this script creates a Verilog or VHDL wrapper that instantiates HDL modules defined by files in the **hdl** directory. During system generation, a typical generator script copies the HDL files to the Quartus II project directory, and renames the files uniquely to match the instance name. The generator script can also define parameters for the top-level HDL module, based on parameters specified in the SOPC Builder GUI.

☞ The generator script for most Altera-provided components dynamically generates HDL for each instance of the component, based on parameters specified in the SOPC Builder GUI. Such components do not include HDL files, because the HDL is generated programmatically.

hdl Directory

This subdirectory contains HDL files that describes the component hardware. This directory is not required to exist, and the name of the directory is not significant. The generator script specifies the exact path to HDL files, if any.

Other Component Files

The component directory can contain other files and subdirectories, depending on the component's design and recommended usage. Typically, such items are not used directly by SOPC Builder, but are necessary for other stages of development. SOPC Builder ignores any files it cannot identify for its own purposes.

Items you might find in a component directory include the following:

- **inc** subdirectory – This directory includes the register map for a peripheral component. The register map is typically declared as a C header file that defines symbols and macros for accessing the component hardware. SOPC Builder does not use this information directly. For Nios II processor systems, the Nios II IDE uses the contents of **inc**.
- **HAL** subdirectory – This directory contains software drivers for a component. SOPC Builder does not use this information directly. For Nios II processor systems, the Nios II IDE uses the contents of **HAL** to generate software for the component.
- **UCOSII** subdirectory – This directory contains software drivers for a component, specific to the MicroC/OS-II real-time operating system (RTOS). SOPC Builder does not use this information directly. For Nios II processor systems, the Nios II IDE uses the contents of **UCOSII** to generate software for the component.
- Files associated with the generator script – The generator script may require data files, Perl libraries, or other files.
- **sdk** subdirectory – This directory contains software and header files to support the legacy software development kit (SDK) flow for the first-generation Nios processor.
- Data files, source code, or other files needed by tools other than SOPC Builder

Component Directory Location

Each time SOPC Builder starts up, it looks for component directories. Any components that it finds are displayed in the list of available components on the SOPC Builder **System Contents** tab.

SOPC Builder searches the following locations for component directories:

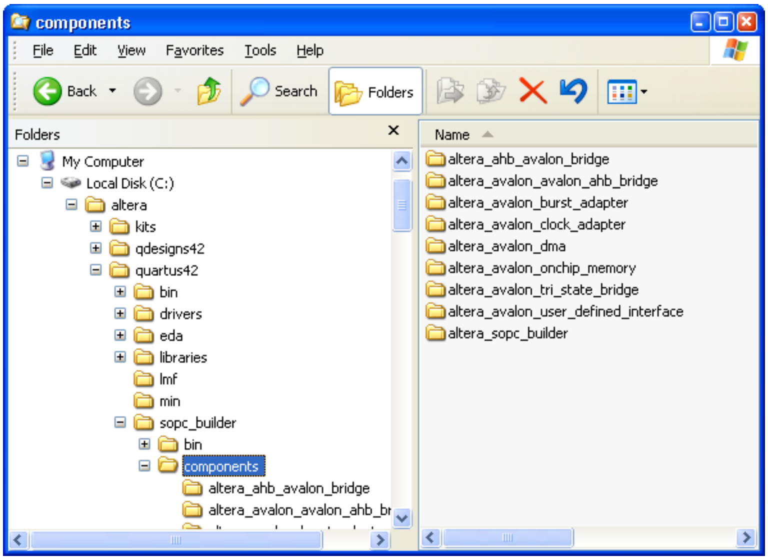
1. The current Quartus II project directory
2. Any directories specified under **Component/Kit Library Search Path** on the SOPC Builder Setup page in the SOPC Builder GUI.
3. **SOPC_BUILDER_PATH**
4. **QUARTUS_ROOTDIR/sopc_builder/**

The following describes the process by which SOPC Builder identifies components:

- **SOPC_BUILDER_PATH** and **QUARTUS_ROOTDIR** are environment variables that are defined during the Quartus II installation process.
- SOPC Builder searches each of these locations for subdirectories in the order shown, and then searches each subdirectory for a **class.ptf** file.
- When SOPC Builder finds a **class.ptf** file, it reads the file, identifies the component, and makes the component available in the SOPC Builder GUI.
- If multiple instances of the same component class exist, SOPC Builder uses the following rules to determine which one to use:
  - The component with the highest version takes precedence.
  - If all component versions are equal, the first component found takes precedence.
- If a directory is recognized as a kit directory (indicated by the presence of a file named **.sopc_builder**), then SOPC Builder further searches in the **components/** subdirectory.

Figure 4–3 shows an example of the directory of components installed with the Quartus II software. Note that not all directories correspond directly to user-visible components.

Figure 4–3. Avalon Component Directories



Folders

- My Computer
  - Local Disk (C:)
    - altera
      - kits
      - qdesigns42
      - quartus42
        - bin
        - drivers
        - eda
        - libraries
        - lmf
        - min
        - sopc_builder
          - bin
          - components
            - altera_ahb_avalon_bridge
            - altera_avalon_avalon_ahb_br

Name

altera_ahb_avalon_bridge
altera_avalon_avalon_ahb_bridge
altera_avalon_burst_adapter
altera_avalon_clock_adapter
altera_avalon_dma
altera_avalon_onchip_memory
altera_avalon_tri_state_bridge
altera_avalon_user_defined_interface
altera_sopc_builder

5. Component Editor

QII54005-6.0.0

Introduction

This chapter describes the SOPC Builder component editor. The component editor is a feature of SOPC Builder that lets you create and edit your own SOPC Builder components. You use the component editor GUI to do the following:

- Import hardware description language (HDL) files (if any) that describe the component hardware.
- Specify the hardware interface(s) to the component.
- Package software drivers into the component directory.
- Declare any parameters that alter the component structure or functionality, and define a user interface to let users parameterize instances of the component.

A typical development sequence might include the following steps, not necessarily in this order:

1. Design and test custom hardware in Verilog or VHDL.

2. Build an SOPC Builder system.

3. Use the component editor to package the custom HDL into an SOPC Builder component.

4. Incorporate one or more instances of your custom component into the SOPC Builder system.

5. Test the system including the new component, and make iterative refinements to the HDL.

6. Use component editor to update the component files.

You can also use the component editor to define an Avalon® interface to logic external to the system module. In this case, you do not provide HDL files, and use the component editor only to define the hardware interface.

Component Editor Output

Based on the settings you make using the component editor GUI, the component editor outputs a properly-formed component directory, containing the following items:

- **class.ptf** file – This file identifies the component, defines how the component hardware connects to the overall system, and declares user-configurable parameters.
- **cb_generator.pl** file – This file is a script used by SOPC Builder during system generation to generate instances of the component hardware.
- **hdl** directory – If the component includes logic inside the system module, this directory contains the HDL files.
- Software files – If this component is a processor peripheral, you can associate software files, such as drivers.

Exiting the component editor automatically prompts you to save the component files. On the File menu you can also click **Save** to save the files. Note that "saving" actually creates or copies multiple files, and stores them in their appropriate places in a component directory.

If you later edit the source code, you must update the component. Refer to section "Re-Editing a Component" on page 5–13.

The component editor always saves your new component to a subdirectory in the current Quartus® II project directory. You can relocate the component directory later, if you wish. For example, you could locate the component files in a central location so that other users can instantiate the component in their systems.

The component editor creates components with the following hardware characteristics:

- A component has one or more interfaces. Typically, an *interface* means an Avalon master port or slave port. The component editor lets you build a component with any combination of Avalon master or slave ports.
- Each interface is comprised of one or more signals.
- The component can use a set of global signals that are not associated to a specific Avalon interface.
- The component can include logic inside the system module, or serve as an interface to logic external to the system module.

See the *SOPC Builder Components* chapter in Volume 4 of the *Quartus II Handbook* for details on components. See the *Avalon Interface Specification* for details on the Avalon interface.

Starting the Component Editor

To start the component editor, in the SOPC Builder GUI on the File menu click **New Component**. When the component editor starts, it displays the **Introduction** tab, which provides a simple introduction to using the component editor.

The component editor GUI presents several tabs that group like settings on the same tab. A message window at the bottom of the component editor displays warning and error messages.

☞ Each tab in the component editor GUI provides on-screen information that describes how to use each tab. Click the triangle at the top-left of each tab to view these instructions.

In general, you will proceed through the tabs from left to right as you progress through the component creation process. You can return to an earlier tab at any time.

HDL Files Tab

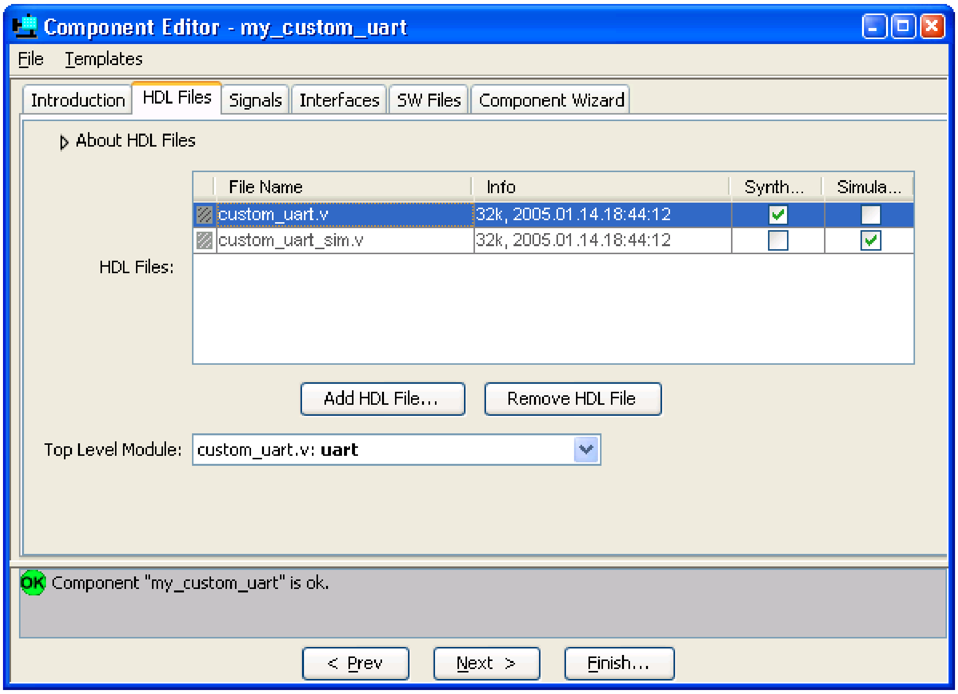
You use the **HDL Files** tab to import existing Verilog or VHDL files that describe the component hardware. When you save the component later, the component editor copies these files into your new component's directory.

☞ If your component is an interface to external logic, then do not specify any files on this tab.

As you add each HDL file, the component editor analyzes the file by invoking the Quartus II Analyzer in the background. If there are errors, a dialog appears on the screen describing the problems. If the file is successfully analyzed, then the component editor identifies any design modules described in the file, and lists them in the **Top Level Module** list. If your HDL contains more than one module, you must select the appropriate top level module in the **Top Level Module** list.

Figure 5–1 shows an example of the **HDL Files** tab.

Figure 5–1. HDL Files Tab



File Name	Info	Synth...	Simula...
custom_uart.v	32k, 2005.01.14.18:44:12	✓	
custom_uart_sim.v	32k, 2005.01.14.18:44:12		✓

Top Level Module: custom_uart.v: **uart**

Component "my_custom_uart" is ok.

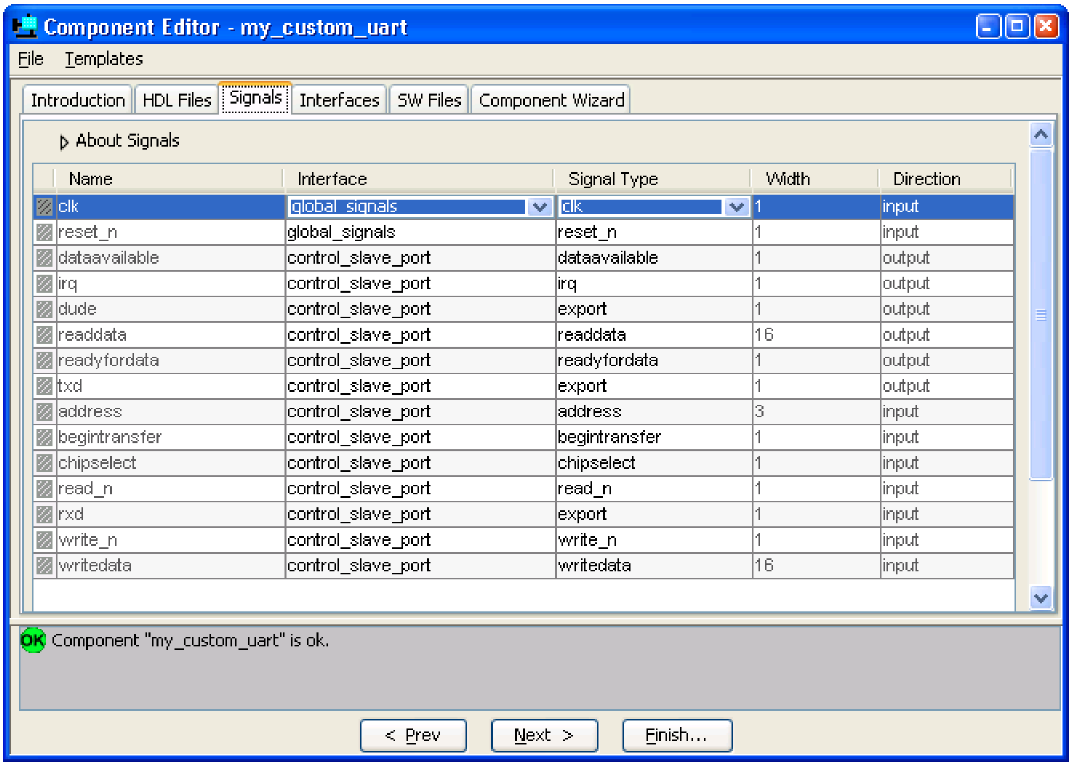
If separate HDL files define the synthesizable hardware and simulation model for the component, use the **Synthesis** and **Simulation** boxes to specify the role for each file.

Signals Tab

You use the **Signals** tab to specify the purpose of each signal in the top level component module. If you specified files on the **HDL Files** tab, then the signals on the top-level module appear on the **Signals** tab. If the component is an interface to external logic, then you must manually add the signals that comprise the interface to the external logic.

Figure 5–2 shows an example of the **Signals** tab.

Figure 5–2. Signals Tab



Name	Interface	Signal Type	Width	Direction
clk	global_signals	clk	1	input
reset_n	global_signals	reset_n	1	input
dataavailable	control_slave_port	dataavailable	1	output
irq	control_slave_port	irq	1	output
dude	control_slave_port	export	1	output
readdata	control_slave_port	readdata	16	output
readyfordata	control_slave_port	readyfordata	1	output
txd	control_slave_port	export	1	output
address	control_slave_port	address	3	input
begintransfer	control_slave_port	begintransfer	1	input
chipselect	control_slave_port	chipselect	1	input
read_n	control_slave_port	read_n	1	input
rxd	control_slave_port	export	1	input
write_n	control_slave_port	write_n	1	input
writedata	control_slave_port	writedata	16	input

OK Component "my_custom_uart" is ok.

Each signal must be assigned a signal type. The default 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** column. In addition to Avalon port interfaces that you create, every component has a *global interface* for common signals such as `clk` and `reset`, and for exported signals.

Naming Signals for Automatic Type and Interface Recognition

The component editor can automatically recognize signal types and interfaces based on the names of signals in the source HDL file. The component editor recognizes signal names that follow a specific structure. Table 5–1 lists the signal name structures.

Table 5–1. Structure of Automatically Recognized Signal Names

Type of Signal	Name Structure
Avalon signal associated with a specific Avalon interface	<*Interface Type*>_<*Interface Name*>_<*Avalon Signal Type*>[`_n`]
Export signal associated with a specific Avalon interface	<*Interface Type*>_<*Interface Name*>`_export_`<*Name*>[`_n`]
Global clock or reset	gls_clk[_n] or gls_reset[_n]

For any value of *Interface Name* the component editor will automatically create an interface by that name, if necessary, and assign the signal to it. *Avalon Signal Type* must match one of the valid Avalon signal types. You can append `_n` to indicate an active-low signal. Table 5–2 lists the valid values for *Interface Type*.

Table 5–2. Valid Values for <Interface Type>

Value	Meaning
avs	Avalon slave port
avm	Avalon master port
ats	Avalon tristate slave port
gls	Global signals not associated to a specific Avalon port

The following example shows a Verilog HDL module declaration with signal names that infer two Avalon slave ports.

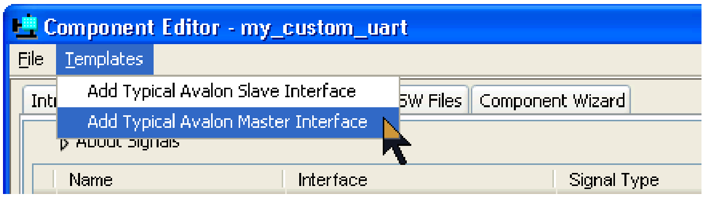
Example: Verilog Module With Automatically Recognized Signal Names

```
module my_multiport_component (
       // Signals for Avalon 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 slave port "s2"
       avs_s2_address,
       avs_s2_read,
       avs_s2_readdata,
       avs_s2_export_dac_output,

       // Global interface siganals
       gls_clk
    );
```

Templates for Interfaces to External Logic

If you are creating an interface to external logic, you can use the **Templates** menu to add a set of signals for a typical Avalon master or slave port. After adding a template, you can add or delete signals to customize the interface to meet your needs. Figures 5–3 shows the **Templates** menu.

Figure 5–3. Templates Menu


Component Editor - my_custom_uart
File
Templates
Add Typical Avalon Slave Interface
Add Typical Avalon Master Interface
Intı
SW Files
Component Wizard
About Signals
Name
Interface
Signal Type

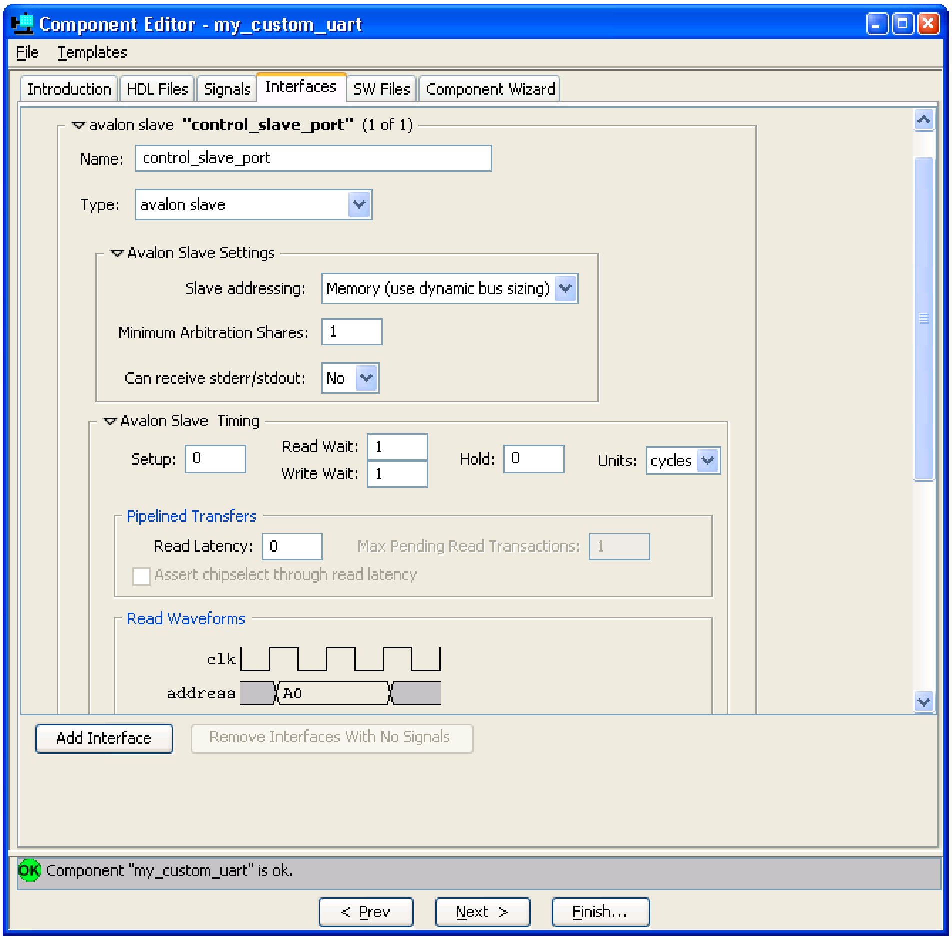
Interfaces Tab

The **Interfaces** tab lets you configure the interfaces on your component, and specify a name for each interface. The interface name identifies the interface, and appears in the SOPC Builder GUI 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 lets you configure the type and properties of each interface. For example, an Avalon slave interface has timing parameters which you must set appropriately.

Figure 5–4 shows an example of the **Interfaces** tab.

Figure 5–4. Interfaces Tab



avalon slave **"control_slave_port"** (1 of 1)

Name: control_slave_port

Type: avalon slave

Avalon Slave Settings

Slave addressing: Memory (use dynamic bus sizing)

Minimum Arbitration Shares: 1

Can receive stderr/stdout: No

Avalon Slave Timing

Setup: 0 Read Wait: 1 Write Wait: 1 Hold: 0 Units: cycles

Pipelined Transfers

Read Latency: 0 Max Pending Read Transactions: 1

Assert chipselect through read latency

Read Waveforms

Component "my_custom_uart" is ok.

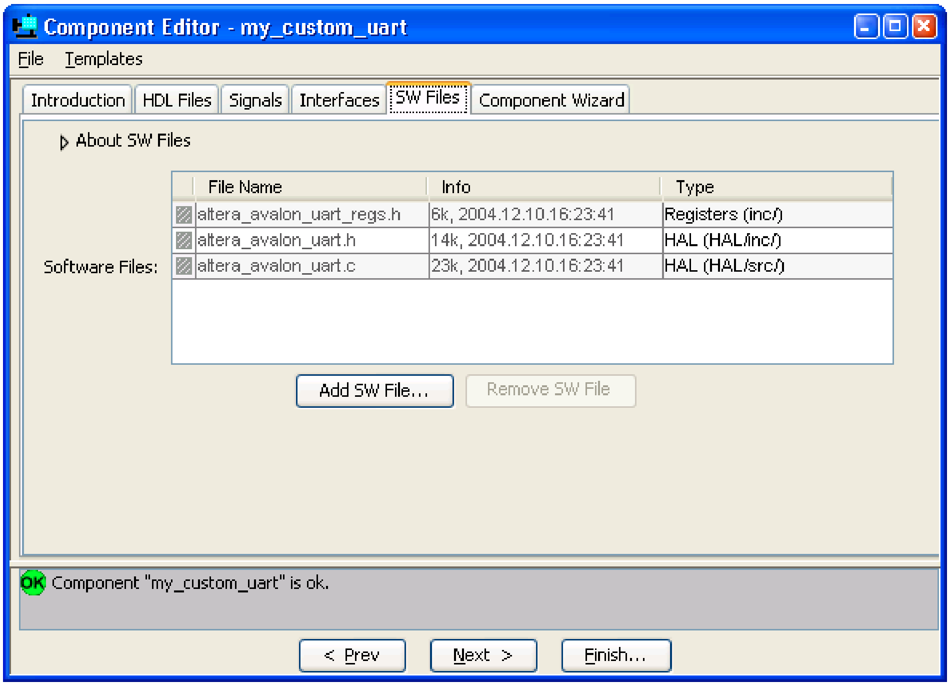
SW Files Tab

The software files tab (**SW Files**) lets you add existing driver software for your new component. When you save the component later, the component editor copies the software files to software subdirectories inside the component directory. The component editor uses the software directory structure specified by the hardware abstraction layer (HAL) for the Nios® II processor.

For information on writing component driver software for the Nios® II processor, including the directory structure, see the *Nios II Software Developer's Handbook*.

Figure 5–5 shows an example of the **SW Files** tab.

Figure 5–5. Software Files (SW Files) Tab



	File Name	Info	Type
▨	altera_avalon_uart_regs.h	6k, 2004.12.10.16:23:41	Registers (inc/)
▨	altera_avalon_uart.h	14k, 2004.12.10.16:23:41	HAL (HAL/inc/)
▨	altera_avalon_uart.c	23k, 2004.12.10.16:23:41	HAL (HAL/src/)

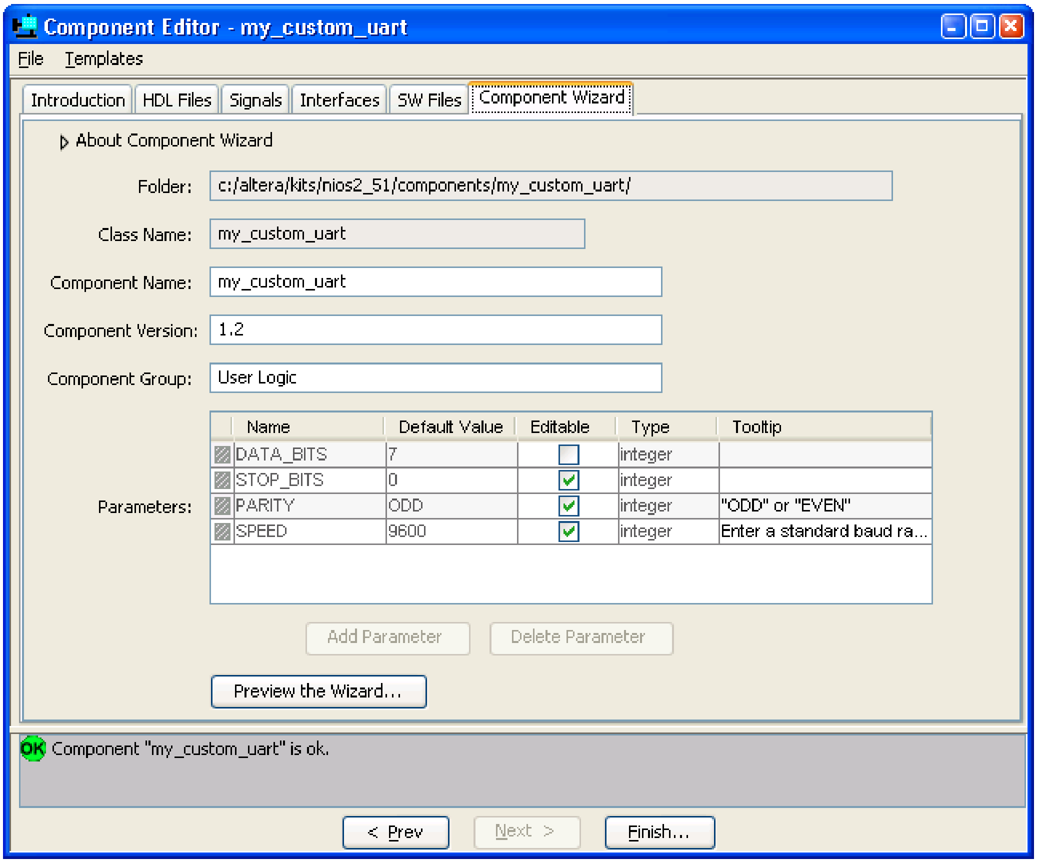
OK Component "my_custom_uart" is ok.

Component Wizard Tab

The **Component Wizard** tab provides settings that affect the presentation of your new component to the user.

Figure 5–6 shows an example of the **Component Wizard** tab.

Figure 5–6. Component Wizard Tab



Folder: c:/altera/kits/nios2_51/components/my_custom_uart/

Class Name: my_custom_uart

Component Name: my_custom_uart

Component Version: 1.2

Component Group: User Logic

Parameters:

Name	Default Value	Editable	Type	Tooltip
DATA_BITS	7	☐	integer	
STOP_BITS	0	✓	integer	
PARITY	ODD	✓	integer	"ODD" or "EVEN"
SPEED	9600	✓	integer	Enter a standard baud ra...

OK Component "my_custom_uart" is ok.

Identifying Information

You can specify information that identifies the component, such as the component name, version, and group. The component name is the name that appears in the list of available components in the SOPC Builder GUI. The name can include spaces or other special characters.

Based on the component name you specify, the component editor creates a valid class name. The class name determines the component directory name.

You can also specify a component version and a component group. The group setting determines in which group SOPC Builder displays your component in the list of available components. If you enter a previously unused group name, SOPC Builder will display a new group by that name.

Parameters

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

If the top-level module of the component HDL declares any parameters (*parameters* for Verilog, or *generics* for VHDL) then those parameters appear in the **Parameters** table. These parameters are presented to users when they create or edit an instance of your component.

☞ To provide parameterization for a lower-level submodule, the top-level module must declare the parameter, and pass it down to the appropriate submodule.

Using the **Parameters** table, you can specify whether or not each parameter is user-editable or not. You can also specify a tooltip that is displayed when a user mouses over the parameter name, to help explain its use.

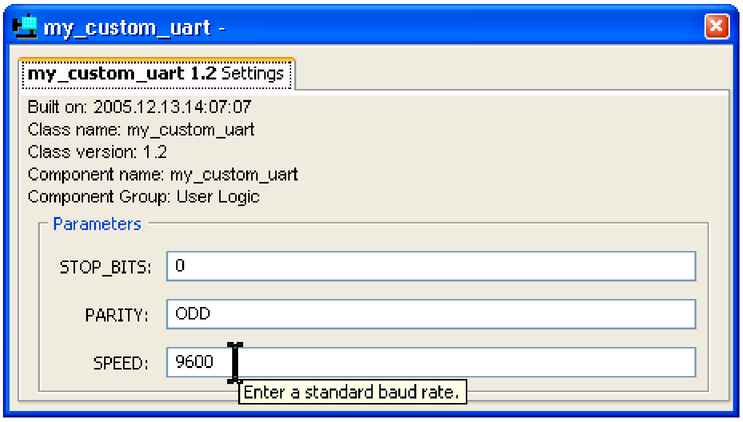
☞ Tooltips can use basic HTML tags, such as `<b>` and `<p>` to format tooltip text. See Figure 5–7.

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 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 look to an end user. Figure 5–7 shows an example of a component wizard preview.

Figure 5–7. Example Component Wizard Preview



Built on: 2005.12.13.14:07:07
Class name: my_custom_uart
Class version: 1.2
Component name: my_custom_uart
Component Group: User Logic

STOP_BITS: 0
PARITY: ODD
SPEED: 9600

Enter a standard baud rate.

Saving a Component

You can save the component by clicking **Finish** on any of the tabs, or by clicking **Save** on the File menu. The component editor saves the component files to the current Quartus® II project directory, in a subdirectory named the same as the component class name specified on the **Component Wizard** tab.

If you later edit the source code, you must update the component. Refer to section "Re-Editing a Component" on page 5–13.

When your component design is final, you can move the component files to a different location.

Re-Editing a Component

After you save a component and exit the component editor, you can re-edit a component at any time from the SOPC Builder GUI. To edit a component, right-click it in the list of available components, and choose **Edit Component**. You cannot edit components that were not created by the component editor, such as Altera®-provided components.

If you alter the source HDL, you need to use the component editor to incorporate the HDL changes into the SOPC Builder component. If you alter the signals on the top-level module, you need to reassign interface signals. If you change the name of a component while editing it, the component editor will save the component files to a new directory.

☞ Existing SOPC Builder systems that use a previous version of the component require further action to reflect the component changes.

To update existing SOPC Builder systems with the new component changes, perform the following steps:

1. Delete any existing instances of the component in the SOPC Builder system, and add the component again.

2. Regenerate the system.

6. Archiving SOPC Builder Projects

QII54017-6.0.0

Introduction

The purpose of this chapter is to help you identify the files you need to include when archiving an SOPC Builder system module. With this information, you can archive:

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

You might need 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 need to decide what source control or archiving tool to use, and you need to 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 must be included in an archived SOPC Builder design.
- Which files must have write permission to allow the design to be generated and the software projects compiled.

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® 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 Development 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 from 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 a system, both the FPGA configuration image (.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 6–1 are located in the Quartus II project directory.

Table 6–1. Files Required for an SOPC Builder System (Part 1 of 2)

File description	File name	Write permission required? *(1)*
SOPC Builder system description	<sopc_builder_system>.ptf	Yes
All non-generated HDL source files *(2)*	e.g. top_level_schematic.bdf, customlogic.v	No

Table 6–1. Files Required for an SOPC Builder System (Part 2 of 2)

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

Notes to Table 6–1:
(1) For further information about write permissions, see "File Write Permissions" on page 6–4.
(2) Include all HDL source files not generated by SOPC Builder. This includes HDL source files you create or copy from elsewhere. To identify a file generated by SOPC Builder, open the file and look for the Altera® header: `Legal Notice: (C)2006 Altera Corporation. All rights reserved.`

Nios II Application Software Project Files

The files listed in Table 6–2 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.*

Table 6–2. Files Required for a Nios II Application Software Project

File description	File name	Write permission required? (1)
All source files	e.g. app.c, header.h, assembly.s, lookuptable.dat	No
Eclipse project file	.project	No
C/C++ Development Toolkit project file	.cdtproject	Yes
C/C++ Development Toolkit option file	.cdtbuild	No
Software configuration file	application.stf	No

Note to Table 6–1:
(1) For further information about write permissions, see "File Write Permissions" on page 6–4.

Nios II System Library Project

The files listed in Table 6–3 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.*

Table 6–3. Files Required for a Nios II System Library Project

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

Note for Table 6–3:
(1) For further information about write permissions, see "File Write Permissions" on page 6–4.

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 need to override this behavior. You do not need to check the files out of source control unless you are modifying the SOPC Builder design or Nios II software project.

7. Board Description Editor

QII54015-6.0.0

Introduction

This chapter introduces SOPC Builder board descriptions and provides complete reference for the board description editor.

Board Descriptions

SOPC Builder board descriptions contain detail about a printed circuit board (PCB) with a mounted Altera® FPGA. Board descriptions encapsulate low-level details about a board that can be reused by all users of the board. A PCB designer creates a board description so that other designers do not have to be board experts to create SOPC Builder systems for the target board. A board description enables designers to create an SOPC Builder design that works in hardware, even though they might not know how to read a board schematic or make Quartus® II pin assignments.

You can obtain board descriptions from several sources:

1. Altera provides ready-made board descriptions for Altera development boards. If you are targeting an Altera development board, use the board description provided with the board.

2. If you are designing an SOPC Builder system for an existing board, you might receive the board description from the board designer as part of the supporting files for the board.

3. If you are a board designer, you create a board description for your custom boards using the SOPC Builder board description editor.

The figures in this document are based on Altera's Nios Development Board, Cyclone™ II Edition as an example. The development tools for the Nios II processor include several ready-made board descriptions which you can use as reference. You can download an evaluation of the Nios II development tools free at **www.altera.com**.

Uses for Board Descriptions

Board descriptions serve three purposes:

- Provide FPGA pinout details to the SOPC Builder pin mapper.
- Define a system template for the target board. A system template is a ready-made SOPC Builder system for the board, which provides a starting place for future users of the board.
- Provide the Nios II flash programmer with details about flash memory on the board.

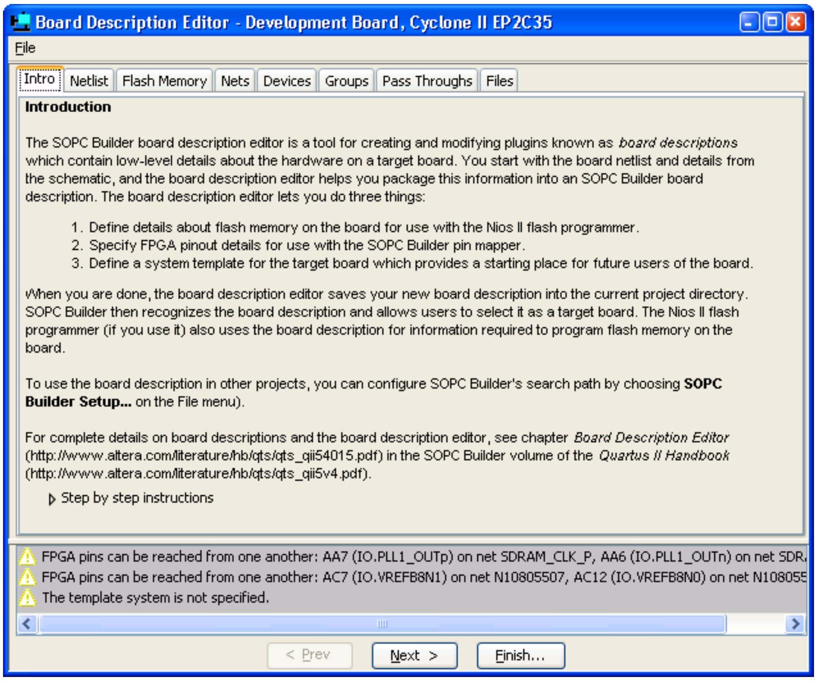
Not all board descriptions serve all three purposes. For example, a board description might contain only information for the Nios II flash programmer. Alternately, it might provide only pinout details and a system template.

For details on the pin mapper, see the chapter *Pin Mapper* in volume 4 of the *Quartus II Handbook*. For details on the Nios II flash programmer, see the *Nios II Flash Programmer User Guide*.

Board Description Editor

The board description editor is a feature of SOPC Builder that provides a graphical user interface (GUI) for creating board descriptions (Figure 7–1). Starting with the PCB netlist and details from the schematic, a PCB designer uses the board description editor to package this information into an SOPC Builder board description.

Figure 7–1. Board Description Editor



Introduction

The SOPC Builder board description editor is a tool for creating and modifying plugins known as *board descriptions* which contain low-level details about the hardware on a target board. You start with the board netlist and details from the schematic, and the board description editor helps you package this information into an SOPC Builder board description. The board description editor lets you do three things:

1. Define details about flash memory on the board for use with the Nios II flash programmer.
2. Specify FPGA pinout details for use with the SOPC Builder pin mapper.
3. Define a system template for the target board which provides a starting place for future users of the board.

When you are done, the board description editor saves your new board description into the current project directory. SOPC Builder then recognizes the board description and allows users to select it as a target board. The Nios II flash programmer (if you use it) also uses the board description for information required to program flash memory on the board.

To use the board description in other projects, you can configure SOPC Builder's search path by choosing **SOPC Builder Setup...** on the File menu).

For complete details on board descriptions and the board description editor, see chapter *Board Description Editor* (http://www.altera.com/literature/hb/qts/qts_qii54015.pdf) in the SOPC Builder volume of the *Quartus II Handbook* (http://www.altera.com/literature/hb/qts/qts_qii5v4.pdf).

Step by step instructions

FPGA pins can be reached from one another: AA7 (IO.PLL1_OUTp) on net SDRAM_CLK_P, AA6 (IO.PLL1_OUTn) on net SDR

FPGA pins can be reached from one another: AC7 (IO.VREFB8N1) on net N10805507, AC12 (IO.VREFB8N0) on net N108055

The template system is not specified.

The board description editor GUI consists of several tabs and a message window. You do not have to use the tabs in order. The board description editor consists of the following tabs:

- Intro Tab
- Netlist Tab
- Flash Memory Tab
- Nets Tab
- Devices Tab
- Groups Tab
- Pass Throughs Tab
- Files Tab

Each tab in the board description editor GUI provides on-screen instructions that describe the usage of the tab. Click the triangle at the top-left of each tab to view the on-screen instructions.

The remainder of this document provides complete reference for the board description editor.

Creating a Board Description

The process of creating a board description is independent from creating an SOPC Builder system. You can build a new SOPC Builder system with no target board in mind, or you can create a new board description with no particular target system in mind. However, if you want to create an SOPC Builder targeting a particular board, you first must create a board description.

There are two possible flows for using the board description editor:

1. Pins Flow - Creates a board description for use with the SOPC Builder pin mapper.

2. Flash Flow - Creates a board description for use with the Nios II flash programmer.

Depending on how you intend to use the board description, you can use one or both of the flows. The two flows are independent of each other, and can be performed in either order.

Pins Flow

The purpose of the pins flow is to process the PBC netlist for use by the SOPC Builder pin mapper. The end result is a simplified model of the PCB. After finishing the pins flow, you can use your new board description as the target board in SOPC Builder, and you can use the pin mapper to map pins to the PCB.

For further information on the pin mapper, see chapter *Pin Mapper* in volume 4 of the *Quartus II Handbook*.

Steps for the Pins Flow

Below are the typical steps for the pins flow:

1. Use the **Netlist** tab to select the netlist for the target PCB and specify the target FPGA device.

2. Use the **Devices** and **Nets** tabs to filter devices and nets that are not used by the pin mapper, such as ground and power pins.

3. Use the **Devices** and **Pass Throughs** tabs to define pass-through devices, such as resistors and buffers.

4. Use the **Groups** tab to combine multiple devices into a logical grouping.

5. Use the **Files** tab to name the board description and specify the version.

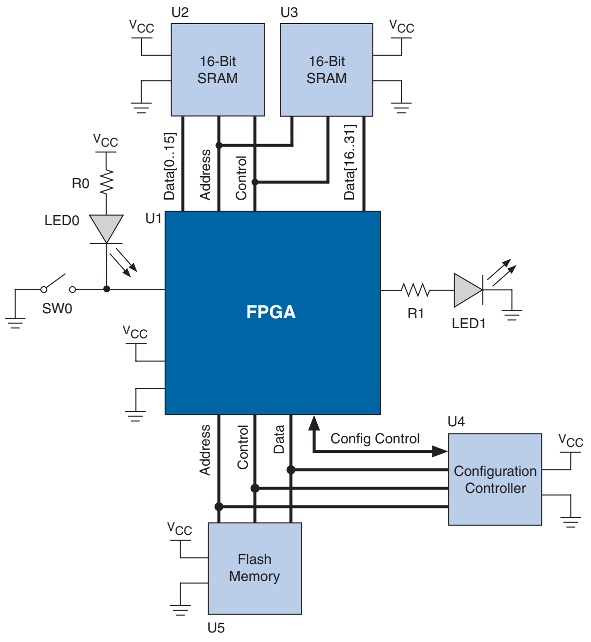
6. Use the **Files** tab to specify a template system for the PCB.

Creating a PCB Model from the Netlist

The pins flow defines the connections between the FPGA and devices on the PCB. You start with the netlist for the PCB and incrementally add details about the board schematic until the board description editor has an appropriate model of the PCB for the pin mapper.

Figure 7–2 shows a board schematic representing a PCB netlist. Without further information, the pin mapper cannot infer how to connect components in an SOPC Builder system to the devices on the PCB.

Figure 7–2. Board Schematic

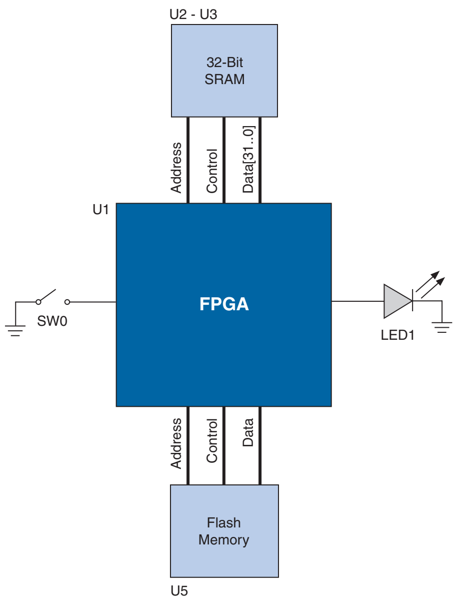


Using Figure 7–2 as an example, you must enter information into the board description editor to overcome the following issues:

- Devices are connected via common GND and VCC nets, but these nets do not carry I/O signals. You must filter these false paths between devices.
- The FPGA drives an LED (LED1), but the net for the FPGA I/O pin connects to a resistor (R1). You must define resistors to be pass-through devices, which are logically transparent devices that I/O signals pass through.
- The SRAM chips (U2, U3) are connected in parallel to be used as a single logical memory. You must define a device group to combine the SRAM chips into a single logical device.
- Certain devices with hard-wired functionality (LED0 and U4) have no relation to the function of the SOPC Builder system in the FPGA. If the SOPC Builder system never needs to connect to a certain device, you can filter the device so that the pin mapper cannot map pins to it.

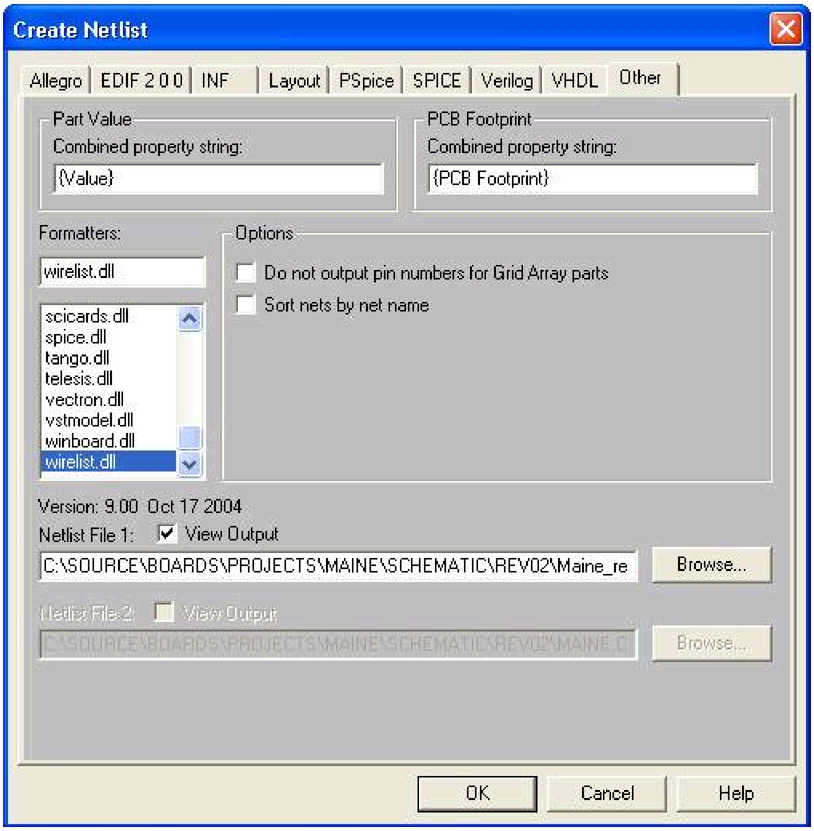
After you provide these details, the pin mapper can model the PCB appropriately, as shown in Figure 7–3.

Figure 7–3. Model of the PCB After Using the Board Description Editor



To begin the pins flow, you need a netlist of the PCB. You must export a netlist in the Wirelist format from your PCB design tool. Figure 7–4 shows an example of the settings in OrCad to create the Wirelist file.

Figure 7–4. Creating a Wirelist File in OrCad



Create Netlist

Part Value
Combined property string: {Value}

PCB Footprint
Combined property string: {PCB Footprint}

Formatters: wirelist.dll

scicards.dll
spice.dll
tango.dll
telesis.dll
vectron.dll
vstmodel.dll
winboard.dll
wirelist.dll

Options
- [ ] Do not output pin numbers for Grid Array parts
- [ ] Sort nets by net name

Version: 9.00 Oct 17 2004

Netlist File 1: [✓] View Output

C:\SOURCE\BOARDS\PROJECTS\MAINE\SCHEMATIC\REV02\Maine_re

Netlist File 2: [ ] View Output

C:\SOURCE\BOARDS\PROJECTS\MAINE\SCHEMATIC\REV02\MAINE.C

The method you use to produce a Wirelist file differs if you use a different PCB layout tool or a different version of OrCad. If you use OrCad, do not use the Convert view on your symbols, because it causes OrCad to generate incorrect Wirelist files.

Flash Flow

The purpose of the flash flow is to define the flash memory devices on the PCB for use by the Nios II flash programmer. After finishing the flash flow, you can use the Nios II flash programmer to program the flash memory on the PCB.

For further information on using the Nios II flash programmer, see the *Nios II Flash Programmer User Guide*.

The flash flow declares the presence of flash memory on the PCB, and defines ranges in these devices for storing FPGA configuration data. Use the flash flow if SOPC Builder systems for this board will include any of the following components:

- Flash Memory (Common Flash Interface)
- EPCS Serial Flash Controller

Below are the typical steps for the flash flow:

1. Use the **Flash Memory** tab to declare the flash memory devices on the board.

2. Use the **Flash Memory** tab to specify regions of flash memory for FPGA configuration data.

Board Description Editor Output

This section describes the files output by the board description editor, and explains how to share board descriptions with other designers.

Board Description File Structure

Structurally, a board description is a set of files recognized by SOPC Builder and the Nios II IDE. Board descriptions use a file structure similar to SOPC Builder component directories. Based on the settings you make in the GUI, the board description editor outputs a properly formed directory, containing the following items:

- *class.ptf file* - Contains defining information about the board description which allows SOPC Builder and the Nios II IDE to recognize it.
- *netlist folder* - Contains the board netlist, if you provide one.
- *system folder* - Contains the system template for the board, if you provide one.

For details on SOPC Builder component file structure, see the chapter *Components* in volume 4 of the *Quartus II Handbook*.

Using Board Descriptions

After you create a board description, you can share it with other designers. To use a board description, you must store the files in a path where SOPC Builder searches for components. To configure SOPC Builder's search path, on the SOPC Builder File menu click **SOPC Builder Setup**.

Starting the Board Description Editor

You can start the board description editor from the SOPC Builder GUI by doing one of the following:

- On the File menu click **New Board Description** to create a new board description.
- On the File menu click **Edit Board Description** to edit the board description for the currently-selected target board.

When the board description editor starts, it displays the **Intro** tab.

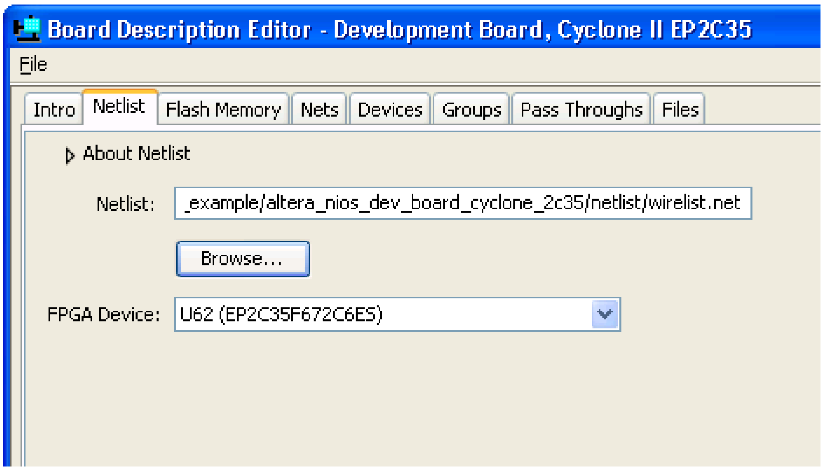
Intro Tab

The **Intro** tab provides a brief description of the board description editor. Figure 7–1 on page 7–3 shows an example of the **Intro** tab.

Netlist Tab

The **Netlist** tab lets you to choose the netlist for the PCB and the target FPGA. Figure 7–5 shows an example of the **Netlist** tab.

Figure 7–5. Netlist Tab



Netlist: _example/altera_nios_dev_board_cyclone_2c35/netlist/wirelist.net

FPGA Device: U62 (EP2C35F672C6ES)

Use the **Netlist** box or the **Browse** button to choose the netlist. The board description editor immediately parses the file for device, pin, and net information.

The netlist must be in the Wirelist format.

After choosing the netlist, you must select which device is the target FPGA in the **Target FPGA** list.

☞ If there are multiple FPGAs on the board that will contain SOPC Builder systems, then you must create multiple board descriptions. Each board description contains pin mapper-related information for one FPGA.

If the netlist contains device names matching the pattern EP1* or EP2* (the first letters of Altera device ordering codes), then the **Target FPGA** list limits your choices to only those devices. Otherwise, it presents all devices in the netlist.

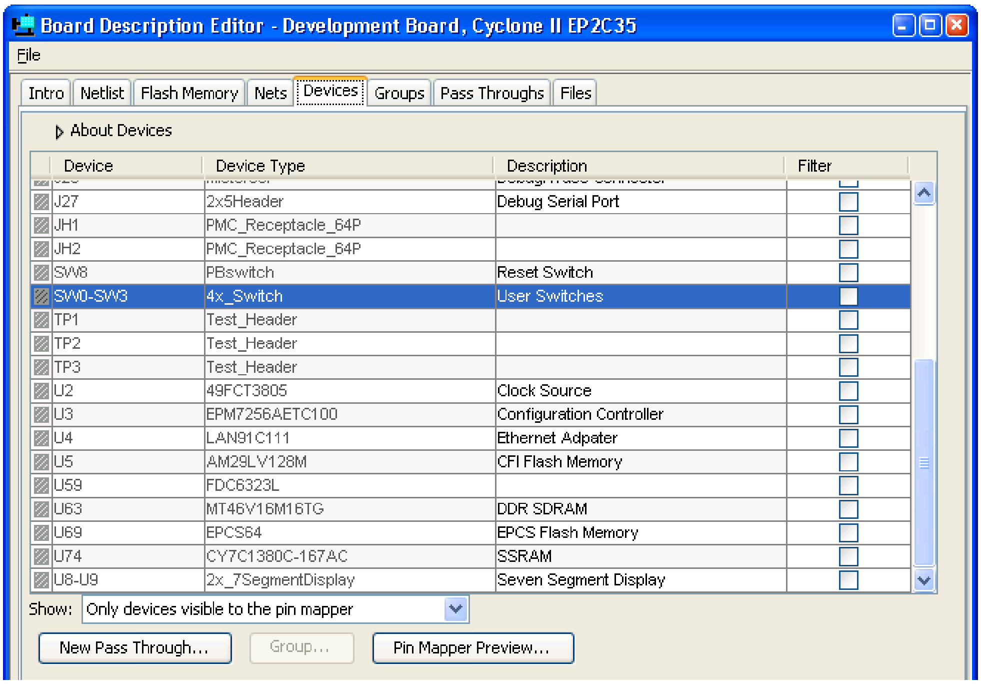
Based on the **Netlist** and **Target FPGA** settings, the board description editor determines possible connections between the FPGA pins and other devices on the PCB.

Devices Tab

The **Devices** tab lets you manage the list of devices that are visible to the pin mapper. Your goal is to manipulate the device list on the **Devices** tab until only devices appropriate for the pin mapper remain. The **Devices** tab is central to the pins flow. For more information, see section "Pins Flow" on page 7–4.

Figure 7–6 shows an example of the **Devices** tab.

Figure 7–6. Devices Tab



Device	Device Type	Description	Filter
J27	2x5Header	Debug Serial Port	☐
JH1	PMC_Receptacle_64P		☐
JH2	PMC_Receptacle_64P		☐
SW8	PBswitch	Reset Switch	☐
SW0-SW3	4x_Switch	User Switches	☐
TP1	Test_Header		☐
TP2	Test_Header		☐
TP3	Test_Header		☐
U2	49FCT3805	Clock Source	☐
U3	EPM7256AETC100	Configuration Controller	☐
U4	LAN91C111	Ethernet Adpater	☐
U5	AM29LV128M	CFI Flash Memory	☐
U59	FDC6323L		☐
U63	MT46V16M16TG	DDR SDRAM	☐
U69	EPCS64	EPCS Flash Memory	☐
U74	CY7C1380C-167AC	SSRAM	☐
U8-U9	2x_7SegmentDisplay	Seven Segment Display	☐

Show: Only devices visible to the pin mapper

Device List

The board description editor populates the device list with the devices found in the netlist. The columns of the device list are described in Table 7–1.

Table 7–1. Device List Columns

Device	Device Type	Description	Filter
The reference designator for each device defined in the netlist file.	Additional device information contained in the netlist file. There is no standard data type or format for **Device Type**. This data is read-only.	Text you type in the **Description** cell appears in the pin mapper, so that future users of the pin mapper easily understand what each device is.	Turning on **Filter** for a device excludes it from the list of devices visible to the pin mapper.

Use the **Show** list to switch between the following modes:

- **All devices** - Lists every device in the netlist.
- **Only devices visible to the pin mapper** - Lists only devices that will be available to the pin mapper after you finish the board description.

All devices listed in the **Only devices visible to the pin mapper** mode will be available as a target device in the pin mapper later.

Filtered Nets, Pass Throughs & Device Groups

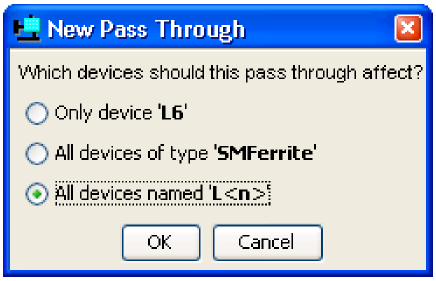
Filtered nets, pass throughs, and device groups modify the way the board description editor displays devices in the device list. You can create new pass throughs and device groups with the **Devices** tab.

☞ The device list changes dynamically based on the settings you make on the **Nets**, **Pass Throughs**, and **Groups** tabs. For further information see the respective sections "Nets Tab", "Pass Throughs Tab", and "Groups Tab".

Creating Pass Throughs

To specify a device as a pass-through device, select it in the device list and click **New Pass Through**. The **New Pass Through** dialog appears, allowing you to choose what type of pass through to create. Figures 7–7 shows an example of creating a pass through for an inductor.

Figure 7–7. Creating a New Pass Through


New Pass Through

Which devices should this pass through affect?

( ) Only device '**L6**'

( ) All devices of type '**SMFerrite**'

[● All devices named 'L<n>']

There are three types of pass throughs:

- *Single device* – Applies only to one specific device, such as a single inductor with reference designator L6.
- *Device type* – Applies to all devices with the same **Device Type**, such as all inductors with **Device Type** "SMFerrite".
- *Reference designator pattern* – Applies to all devices with numbered reference designators sharing a common prefix, such as L*n*, where *n* is any sequence of digits.

After creating a new pass through, you must go to the **Pass Throughs** tab to define the pass through properties.

Creating Device Groups

To specify that multiple devices form a device group, select multiple devices in the device list and click **Group**. To select multiple devices, press and hold the Control key while selecting devices in the list.

After creating a new device group, you must go to the **Groups** tab to name the pins on the device group.

Filtering False Target Devices

Some devices in the netlist file are meaningless to the pin mapper. On the **Devices** tab, you can turn on **Filter** for false target devices to make them invisible to the pin mapper.

Filter the following types of false target devices:

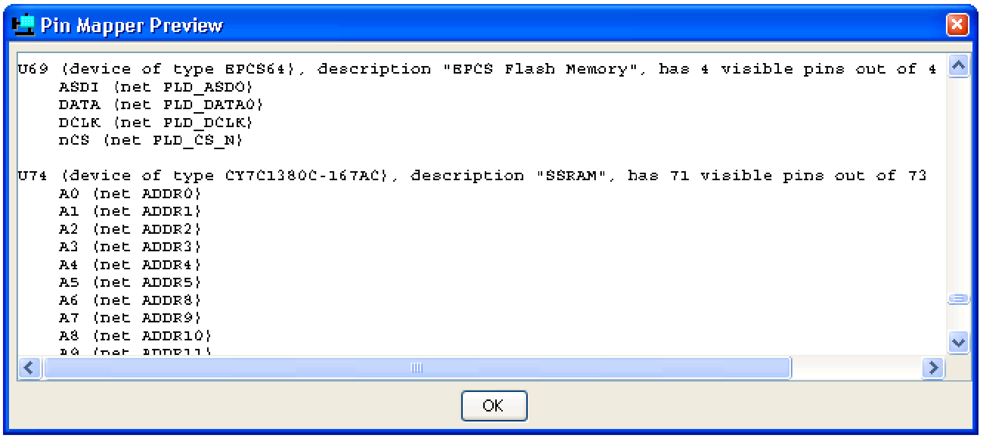
- *Damping capacitors* - Some FPGA pins have a capacitor connected to ground to improve signal integrity. These capacitors are not the intended target for any I/O signal. The board description editor automatically filters any device with reference designator of the form C*n*, where *n* is any sequence of digits.
- *Secondary target devices* - If an FPGA pin connects to two (or more) target devices, you might want to filter the device(s) that do not serve the primary function of the pin. For example, in Figure 7–2 on page 7–6 switch SW0 and LED0 connect to the same FPGA pin. Closing SW0 pulls the FPGA pin low and turns on LED0. In this case, SW0 is the primary function for the FPGA pin, and eliminating LED0 from the pin mapper is a good idea.
- *False paths back to the FPGA* - There might be devices that create false signal paths leading back to the FPGA. For example, termination resistors between differential-pair nets create a loop between FPGA pins that is not a logical signal path.
- *No-stuff devices* - Some devices in the netlist are not stuffed on the PCB during manufacturing. You can filter such devices to disallow pin mapper users from connecting FPGA signals to a target device that does not exist on the board.

☞ If you turn on **Filter** when **Show** is set to **Only devices visible to the pin mapper**, the device immediately disappears from the device list. To see all filtered and unfiltered devices, select **All devices** in the **Show** list.

Previewing Pins Visible to the Pin Mapper

You can click **Pin Mapper Preview** to open the Pin Mapper Preview window which displays the complete list of devices and pins visible to the pin mapper. You are finished with the pins flow when the Pin Mapper Preview window contains exactly the devices and pins you wish to expose to the pin mapper. Figure 7–8 shows an example of the preview listing for a device.

Figure 7–8. Pin Mapper Preview Window



```
U69 (device of type EPCS64), description "EPCS Flash Memory", has 4 visible pins out of 4
    ASDI (net PLD_ASDO)
    DATA (net PLD_DATA0)
    DCLK (net PLD_DCLK)
    nCS (net PLD_CS_N)

U74 (device of type CY7C1380C-167AC), description "SSRAM", has 71 visible pins out of 73
    A0 (net ADDR0)
    A1 (net ADDR1)
    A2 (net ADDR2)
    A3 (net ADDR3)
    A4 (net ADDR4)
    A5 (net ADDR5)
    A6 (net ADDR8)
    A7 (net ADDR9)
    A8 (net ADDR10)
```

For each device with pins visible to the pin mapper, the preview lists the following:

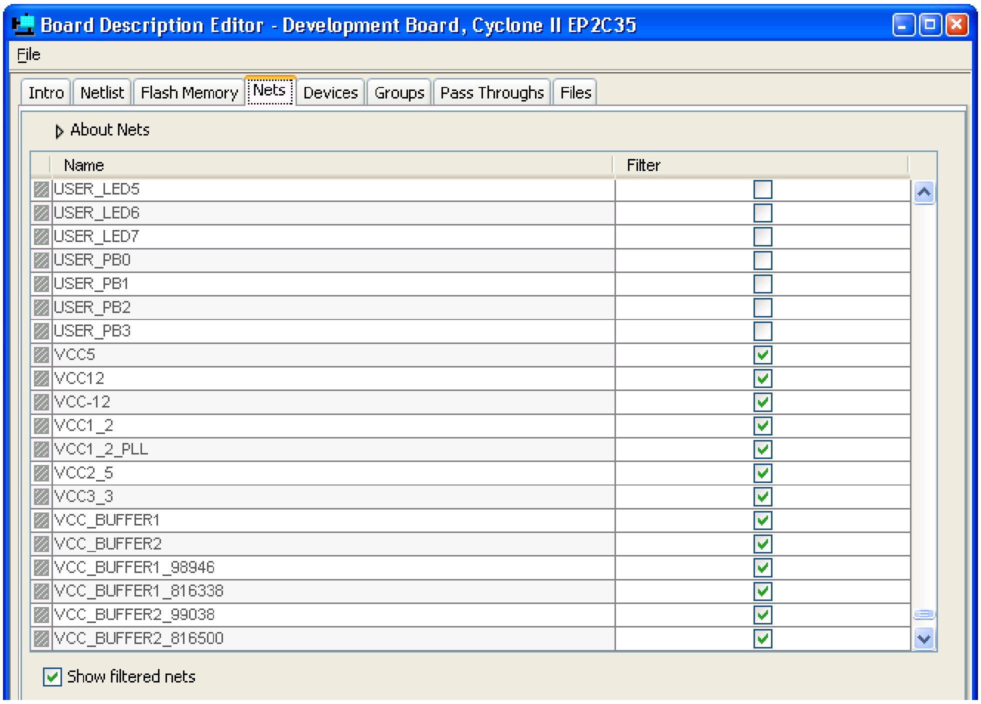
- Reference designator, device type, and description, as shown on the **Devices** tab.
- Total number of pins connected to unfiltered nets, and the number of pins visible to the pin mapper. These numbers provide a useful sanity check. For example, imagine an eight-pin device with one power pin and one ground pin. If power and ground nets are filtered, the total number of unfiltered pins is six. Of the six total pins, perhaps 4 connect to the FPGA.
- The name of each pin visible to the pin mapper, and its corresponding FPGA pin net name.

Nets Tab

The **Nets** tab lets you manage the list of nets that the board description editor considers as signal paths from device-to-device. Typically, multiple devices connect to common power and ground nets, even though not all of these devices have I/O interconnections. Your goal is to filter the netlist to exclude nets that do not carry I/O signals, such as power and ground nets, so that the pin mapper does not detect false signal paths between devices.

Turn on **Filter** for non-I/O nets. Figure 7–9 shows an example with all power supply nets filtered, which prevents the pin mapper from including these nets as signal paths between devices.

Figure 7–9. Nets Tab



Name	Filter
USER_LED5	☐
USER_LED6	☐
USER_LED7	☐
USER_PB0	☐
USER_PB1	☐
USER_PB2	☐
USER_PB3	☐
VCC5	✓
VCC12	✓
VCC-12	✓
VCC1_2	✓
VCC1_2_PLL	✓
VCC2_5	✓
VCC3_3	✓
VCC_BUFFER1	✓
VCC_BUFFER2	✓
VCC_BUFFER1_98946	✓
VCC_BUFFER1_816338	✓
VCC_BUFFER2_99038	✓
VCC_BUFFER2_816500	✓

[✓] Show filtered nets

The board description editor automatically filters nets matching the pattern GND* or VCC*.

Changes you make on this tab affect the device list on the **Devices** tab. If you filter all nets connecting a device to the FPGA, the **Devices** tab ignores the device.

Pass Throughs Tab

A pass through is a logically transparent device that exists along a signal path but does not alter the signal's logic level, such as a resistor. The **Pass Throughs** tab displays all pass throughs created with the **Devices** tab. After you create a new pass through with the **Devices** tab, you use the **Pass Throughs** tab to define the pin-to-pin paths that are logically transparent. For information on creating pass throughs, see section "Creating Pass Throughs" on page 7–13.

Some FPGA I/O pins drive signals through a pass-through device. Pass throughs are not the signal's final destination. For example, in Figure 7–2 on page 7–6 an FPGA pin connects to R1, but the true signal destination is LED1. You define pass throughs to allow the board description editor to detect the true signal destination beyond the pass through device.

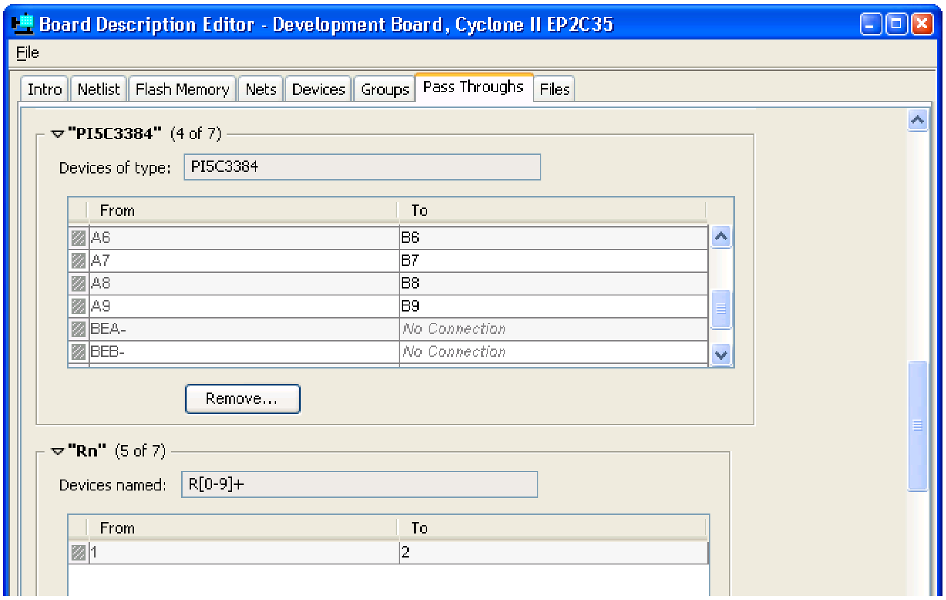
You might have to create pass throughs for the following types of logically transparent devices:

- Resistors in signal paths to limit current flow
- Buffers, level shifters, and power transistors
- Inductors in signal path to improve signal integrity

Figure 7–10 shows an example of the **Pass Throughs** tab with the following pass throughs:

- *For device type PI5C3384* – This pass through affects voltage level-shifters labeled "PI5C3384". Signals pass transparently from pin A*n* to B*n*, where *n* is any digit.
- *For reference designators matching pattern Rn* – This pass through affects all resistors with reference designator R*n,* where *n* is any sequence of digits. Signals pass transparently through the two pins on these resistors.

Figure 7–10. Pass Throughs Tab



"PI5C3384" (4 of 7)

Devices of type: PI5C3384

From	To
A6	B6
A7	B7
A8	B8
A9	B9
BEA-	*No Connection*
BEB-	*No Connection*

"Rn" (5 of 7)

Devices named: R[0-9]+

From	To
1	2

For each entry on the **Pass Throughs** tab, you use the table to specify transparent pin-to-pin paths through the device. For each pin in the **From** list, select an appropriate **To** setting. If a pin does not have a logically transparent path through the device, select **No Connection**. **From** and **To** settings only imply transparent paths through the device; they do not imply signal direction.

Changes you make on this tab affect the device list on the **Devices** tab. After you define a pass through for a particular device, the **Devices** tab no longer displays the device. Instead, it displays the device(s) connected to the other side of the pass-through device.

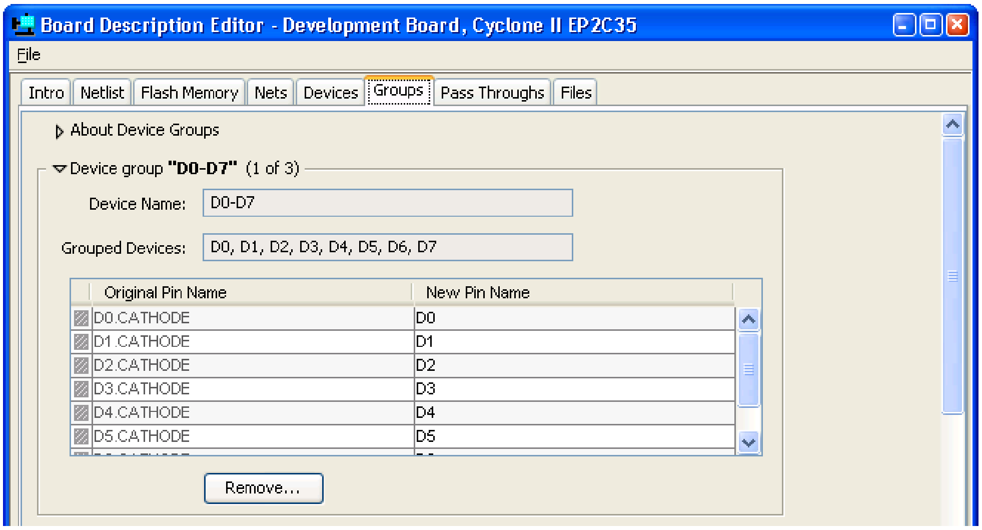
Groups Tab

A device group is comprised of multiple devices that the pin mapper treats as a single device. The **Groups** tab displays all device groups created with the **Devices** tab. After you create a new group with the **Devices** tab, you use the **Groups** tab to name the pins on the group. For information on creating device groups, see section "Creating Device Groups" on page 7–14.

If the PCB is designed for multiple devices to connect to a single SOPC Builder component in the FPGA, creating a device group tends to make the pin mapper easier to use for that component. For example, in Figure 7–2 on page 7–6 grouping the two 16-bit SRAM devices creates a 32-bit SRAM device group.

Figure 7–11 shows the **Groups** tab for the case of eight individual LEDs (D0 to D7) combined to form an 8-bit bank of LEDs.

Figure 7–11. Groups Tab


Board Description Editor - Development Board, Cyclone II EP2C35
File
Intro
Netlist
Flash Memory
Nets
Devices
Groups
Pass Throughs
Files
About Device Groups
Device group "D0-D7" (1 of 3)
Remove...

Device Name: D0-D7

Grouped Devices: D0, D1, D2, D3, D4, D5, D6, D7

Original Pin Name	New Pin Name
D0.CATHODE	D0
D1.CATHODE	D1
D2.CATHODE	D2
D3.CATHODE	D3
D4.CATHODE	D4
D5.CATHODE	D5

For each device group on the **Groups** tab, you use the table to specify a new pin name for each pin that existed on the individual devices. For each pin listed in the **Original Pin Name** list, type a descriptive name in **New Pin Name**. This name appears as the pin name in the pin mapper.

The board description editor automatically fills **New Pin Name** cells with default pin names. However, you must manually inspect all new pin names to verify appropriateness.

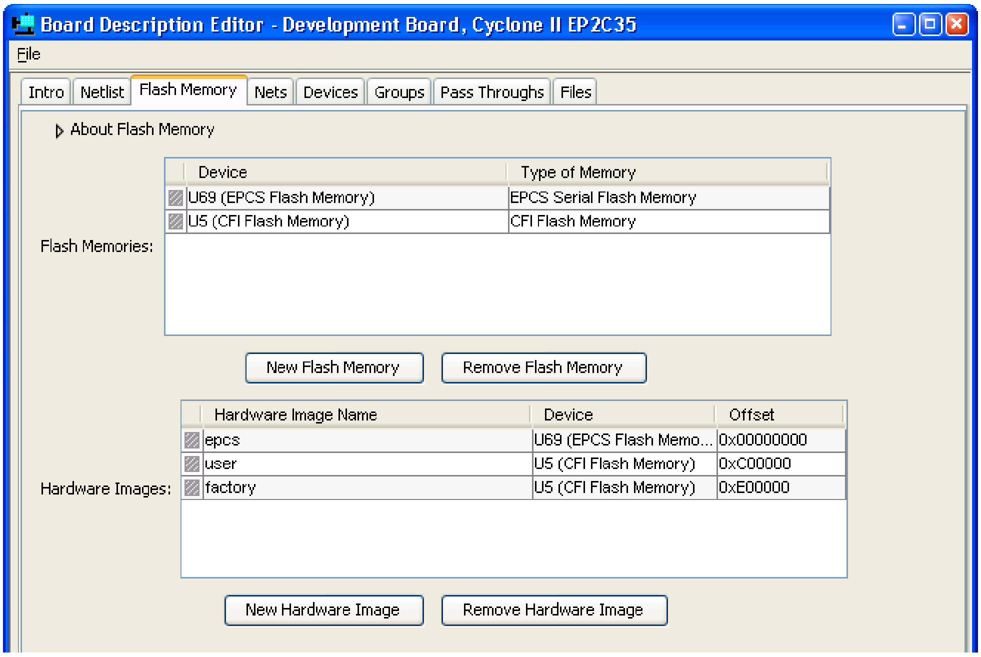
Flash Memory Tab

The **Flash Memory** tab lets you specify details about flash memory on the PCB. This information is used by the Nios II flash programmer. If you do not intend to use the Nios II flash programmer with this board, or if the board does not contain flash memory, then you can ignore this tab. The **Flash Memory** tab is central to the flash flow. For more information, see section "Flash Flow" on page 7–8.

Figure 7–12 shows an example of the **Flash Memory** tab for a PCB with the following flash memories:

- One Altera EPCS serial configuration device containing one hardware image, which is the maximum number of images for an EPCS device.
- One CFI flash memory containing two hardware images. An external configuration controller chip recognizes hardware images at offsets `0xC00000` and `0xE00000` in the flash memory.

Figure 7–12. Flash Memory Tab



Device	Type of Memory
U69 (EPCS Flash Memory)	EPCS Serial Flash Memory
U5 (CFI Flash Memory)	CFI Flash Memory

Hardware Image Name	Device	Offset
epcs	U69 (EPCS Flash Memo...	0x00000000
user	U5 (CFI Flash Memory)	0xC00000
factory	U5 (CFI Flash Memory)	0xE00000

You must add a device entry for each flash memory on the PCB that connects to the FPGA. Click **New Flash Memory** to add devices to the **Flash Memories** list. Select **Type of Memory** for each entry in the **Flash Memories** list.

Each flash memory can contain one or more hardware images, which are address ranges allocated for FPGA configuration data. Define all hardware images stored in flash memory on the PCB.

Click **New Hardware Image** to add an entry to the **Hardware Images** list. For each entry, type a descriptive name into the **Hardware Image Name** box. Use the **Device** list to select which flash memory contains the image. Type the offset where the hardware image data starts into the **Offset** cell.

☞ The first location in the flash memory is offset `0x0000`.

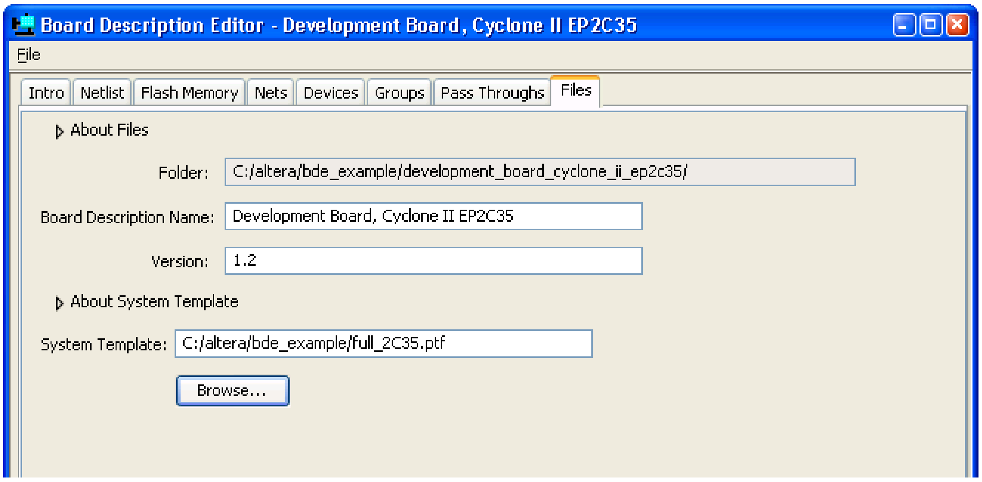
The correct number of hardware images and the offset for each image depend on the following:

- The FPGA configuration controller on the board. In most cases, the configuration controller expects configuration data to exist at specific offsets in flash memory.
- The size of the hardware image data for the target FPGA. The size of the hardware image determines how closely the offsets can be spaced.

Files Tab

The **Files** tab lets you give your new board description a name and version, and specify a system template. Figure 7–13 shows an example of the **Files** tab.

Figure 7–13. Files Tab



Folder: C:/altera/bde_example/development_board_cyclone_ii_ep2c35/

Board Description Name: Development Board, Cyclone II EP2C35

Version: 1.2

System Template: C:/altera/bde_example/full_2C35.ptf

Board Description Name and Version

The name you type in the **Board Description Name** box is the name that appears in the SOPC Builder target board list. This name can include spaces or other special characters. The **Folder** box displays the path where the board description editor saves the board description files. **Board Description Name** determines the **Folder** name. If you are creating a new board description, the editor creates a new folder under the Quartus II project path. If you are editing an existing board description, the editor saves the files back to the same folder, unless you change **Board Description Name**.

Similar to the version for SOPC Builder components, **Version** indicates to SOPC Builder which board description is newer, in the event that more than one of the same kind of board description is detected. SOPC Builder displays only the board description with the highest version in the target board list.

System Template

A system template is a ready-made SOPC Builder system for the board, which provides a starting place for future users of the board. The system template should include:

- An SOPC Builder component corresponding to each device connected to the FPGA.
- Complete pin mappings that connect each component to its target device on the PCB.

To specify the system template, type the path to an SOPC Builder system in the **System Template** box, or click **Browse**.

☞ Do not assign a system template the first time you save your board description. First, use the pin mapper with the new board description to map pins for the system template.

To assign a system template correctly, perform the following steps:

1. In SOPC Builder, open or create the SOPC Builder system you plan to use as the system template.

2. Select the new board description as the target board for the system.

3. Use the pin mapper to assign appropriate pin locations for all signals in the system.

4. Return to the **Files** tab in the board description editor, and select the SOPC Builder system with complete pin mappings as the system template.

5. Re-save the board description.

Saving & Exiting the Board Description Editor

You can save the board description by clicking **Finish** on any of the tabs, or by clicking **Save** on the File menu. Exiting the board description editor automatically prompts you to save the files. The board description editor saves files to a board description folder, creating a new folder if necessary.

☞ If you edit and re-save a board description, and you have existing SOPC Builder systems based on a previous version, you must update those systems to recognize the updates to the board description.

To update an existing SOPC Builder system with board description changes, perform the following steps:

1. Open the system with SOPC Builder.

2. Set the target board to **Unspecified Board**.

3. Set the target board back to the updated board description.

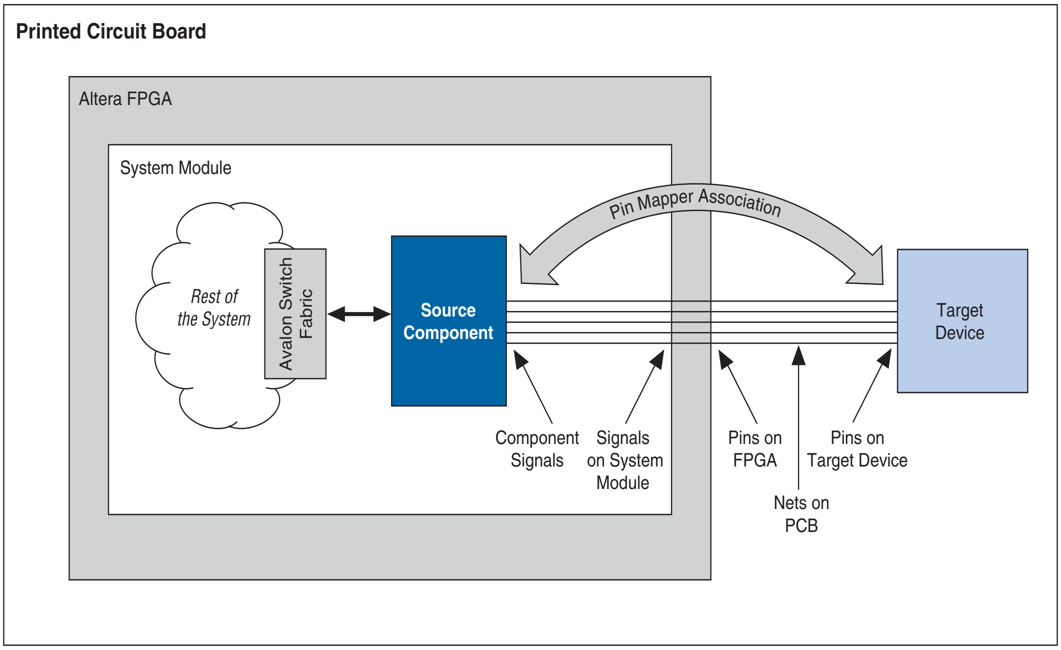
8. Pin Mapper

QII54016-6.0.0

Introduction

The SOPC Builder pin mapper simplifies the process of assigning FPGA pins, allowing you to make device-to-device connections on the printed circuit board (PCB) based on pin function, rather than pin numbers. The pin mapper maps logical connections between SOPC Builder system components and devices mounted on the PCB, as shown in Figures 8–1.

Figure 8–1. Pin Mapper Maps Logical Connections from SOPC Builder Components to Devices on the PCB



The pin mapper accelerates your design time by abstracting details that are tedious and prone to human error, such as:

- Signal names on the SOPC Builder system module
- Pin numbers on the FPGA
- Net names on the PCB

The pin mapper assumes that you know the board you are targeting and that you have an SOPC Builder board description for the target board. Using the board description and information about the components in your SOPC Builder system, the pin mapper GUI lets you map connections from components in the system to device pins on the PCB. To map pins correctly, you must know the function of signals on the source component(s) and pins on the target device(s).

For more information on SOPC Builder board descriptions, refer to the *Board Description Editor* chapter in volume 4 of the *Quartus II Handbook*.

The pin mapper provides a level of assurance that you have accounted for the pin assignments required for a target board. In general, if the Quartus® II project pin assignments do not match the specific board you are using, your design will not function and you could damage the board. The pin mapper helps ensure that you take care of pin assignments for signals on the system module before downloading it to a board.

Design Flow

This section describes the design flow for using the pin mapper, and how the pin mapper affects the Quartus II project.

The following is a typical design flow using the pin mapper:

1. Create an SOPC Builder system targeting a specific board.

2. Instantiate and configure system components on the **System Contents** tab in SOPC Builder.

3. Use the pin mapper to map component signals to devices on the PCB.

4. Generate the system in SOPC Builder. This step automatically applies the pin assignments to the Quartus II project.

   When targeting a specific board, you cannot generate the system until you completely map all signals. Alternately, you can leave all signals unmapped, in which case you must assign all pins using the Quartus II Assignment Editor.

5. Integrate the system module into the top-level Quartus II project.

6. Compile the Quartus II project to fit the design in the FPGA using the pin assignments.

The pin mapper is an optional step in the SOPC Builder system design flow. If you choose to use the pin mapper, then SOPC Builder does not allow you to generate the system until you have completely mapped all signals on the system module.

Applying Pin Assignments to the Quartus II Project

During system generation SOPC Builder generates a Tcl script named <*system module name*>**_setup_quartus.tcl** which contains Quartus II commands to assign pin locations for your system module. At the end of system generation, SOPC Builder executes this script to apply the pin assignments to the Quartus II project.

If you edit the SOPC Builder system later, you can alter the pin mapper settings. However, you must regenerate the system in SOPC Builder to update the Tcl script and apply your pin assignments to the Quartus II project.

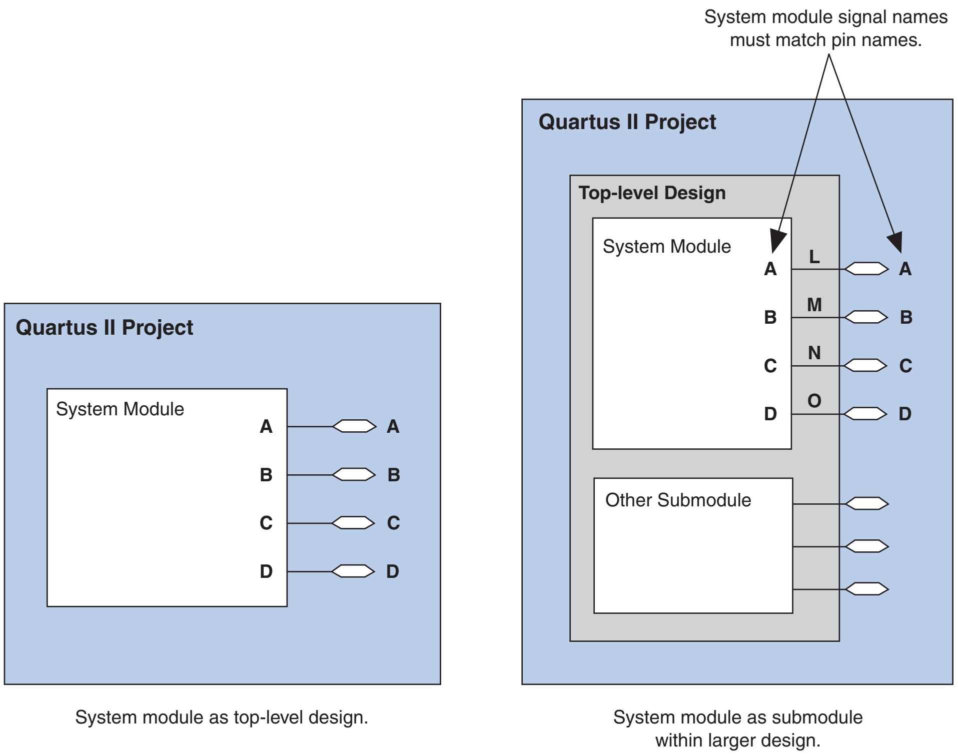
Pin Name Requirements

The <*system module name*>**_setup_quartus.tcl** script makes assignments for pin names matching the names of signals on the system module. Therefore, you must use specific names for pins at the top level of the Quartus II project. For the pin mappings to work, you must name the FPGA pins the same as the signal names on the system module.

For example, imagine a system module with an input vector named `in_port_to_the_button_pio[3..0]`, and suppose that you use the pin mapper to map these signals to four push-button switches on the PCB. The input pins at the top level of the Quartus II project must also be named `in_port_to_the_button_pio[3..0]` for the pin assignments to work.

It does not matter at what level of hierarchy you instantiate the system module in the overall Quartus II project. You can use the system module as the top-level design entity, or you can instantiate the system module inside a lower-level design entity, as shown in Figure 8–2. As long as the top-level pin names match the names of the signals on the system module, the pin mappings will be effective.

Figure 8–2. Pin Mapper and Quartus II Project Hierarchy

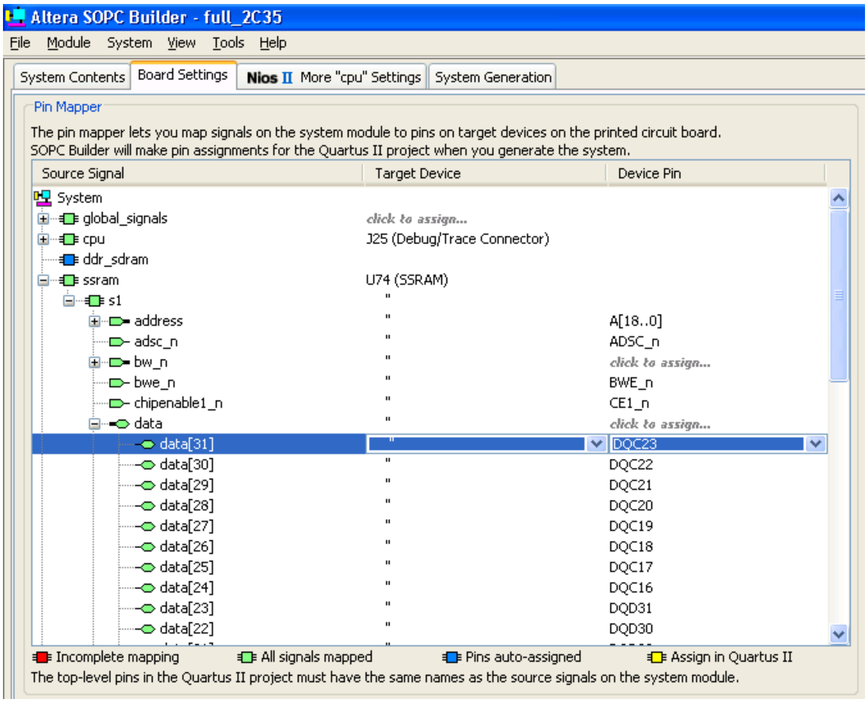

System module signal names must match pin names.
Quartus II Project
System Module
A
B
C
D
A
B
C
D
System module as top-level design.
Quartus II Project
Top-level Design
System Module
A
B
C
D
L
M
N
O
A
B
C
D
Other Submodule
System module as submodule within larger design.

Pin Mapper GUI

The pin mapper GUI is located on the **Board Settings** tab in SOPC Builder, as shown in Figure 8–3.

☞ SOPC Builder displays the **Board Settings** tab only if you specify a target board on the **System Contents** tab. The **Board Settings** tab does not appear when a target board is unspecified, or if the target board description was created prior to version 5.1 of the Quartus II software.

Figure 8–3. Pin Mapper GUI on the Board Settings Tab



Altera SOPC Builder - full_2C35
File Module System View Tools Help
System Contents
Board Settings
Nios II More "cpu" Settings
System Generation
Pin Mapper

The pin mapper lets you map signals on the system module to pins on target devices on the printed circuit board.
SOPC Builder will make pin assignments for the Quartus II project when you generate the system.

Source Signal	Target Device	Device Pin
System		
global_signals	click to assign...	
cpu	J25 (Debug/Trace Connector)	
ddr_sdram		
ssram	U74 (SSRAM)	
s1	"	
address	"	A[18..0]
adsc_n	"	ADSC_n
bw_n	"	click to assign...
bwe_n	"	BWE_n
chipenable1_n	"	CE1_n
data	"	click to assign...
data[31]	"	DQC23
data[30]	"	DQC22
data[29]	"	DQC21
data[28]	"	DQC20
data[27]	"	DQC19
data[26]	"	DQC18
data[25]	"	DQC17
data[24]	"	DQC16
data[23]	"	DQD31
data[22]	"	DQD30

🟥 Incomplete mapping 🟩 All signals mapped 🟦 Pins auto-assigned 🟨 Assign in Quartus II

The top-level pins in the Quartus II project must have the same names as the source signals on the system module.

The pin mapper interface is a table with three columns. Table 8–1 describes the meaning of each column.

Table 8–1. Columns in the Pin Mapper GUI

Source Signals	Target Device	Target Pin
This column contains the names of signals on the components in the system module. This column groups signals to make the list more manageable.	This column contains the names of the devices on the PCB that connect to the FPGA.	This column contains the names of individual pins on the target device. This column shows only the pins available on the selected target device.

Source Signals Column

You can easily browse through source signals on the system module by expanding and collapsing groups in the **Source Signals** column. The signals are grouped together in the following hierarchical levels:

1. *Component level* – Signals for each component are grouped together. In Figure 8–3 on page 8–5, the **ssram** group contains all signals associated with this instance of the component. Component instances with no external signals do not appear in the pin mapper.

2. *Avalon® port level* – Signals that are associated to a particular Avalon port are grouped together. In Figure 8–3 on page 8–5, the **ssram/s1** group contains all signals associated with slave port **s1**. Some signals are not associated with a particular Avalon port and therefore appear outside of a port-level grouping.

3. *Signal vector level* – Vectored signals are grouped together by a common name. In Figure 8–3 on page 8–5, the **ssram/s1/data** group represents a vector of 32 related data signals.

4. *Individual signal level* – The lowest level of hierarchy is the individual signal level. Each individual signal can connect to one FPGA pin, which connects to one or more pins on a target device. In Figure 8–1 on page 8–1, the **ssram/s1/data/data[31]** row represents an individual signal on the system module.

Target Device Column

The **Target Device** column lets you map signals to a target device. You can easily map individual signals, signal vectors, or an entire component to a target device. For example, to map all signals on a component to a specific device, you can highlight a component-level row and specify the device in the **Target Device** column. Alternately, you can individually map each signal to a particular device.

In Figure 8–3 on page 8–5, the component instance **ssram** is mapped to device **U74**. The **Target Device** column shows that lower levels of hierarchy inherit their device settings from the component level.

Depending on the contents of the board description, the pin mapper might have enough information about certain devices on the PCB to automatically map component signals to appropriate pins on the device. In this case, when you target a whole component to a device, the pin mapper automatically maps signals to pins on the target device.

Target Pin Column

The **Target Pin** column lets you map a signal to a specific pin on a device on the PCB. To specify a target pin for a signal, you first must specify **Target Device**. The **Target Pin** column displays only the pins on the target device.

For example, Figure 8–3 on page 8–5 shows that signal `data[31]` on the component instance **ssram** maps to pin DQC23 on device U74.

Vector Signals

You can map a vector signal to a group of pins by clicking the **Target Pin** cell for a vector signal. For a vector signal the **Target Pin** dropdown list displays all groupings of pins on the target device that could logically map to the vector signal.

Differential Signals

To map a signal to pins using a differential I/O standard such as LVDS, you need to map the positive pin only. The Quartus II software will automatically assign the negative pin.

☞ You must set the I/O standard using the Assignment Editor in the Quartus II software.

Signals with Multiple Destinations

Some FPGA I/O pin nets connect to multiple devices on the PCB. As a result, a signal routed to that I/O pin connects to multiple devices. In this case, using the pin mapper to map a signal to any one of the target device pins is equivalent to mapping the signal to all of connected target pins.

Assign in Quartus II

The **Target Pin** column provides the **Assign in Quartus II** setting to make the pin mapper ignore a particular pin. When you set a signal to **Assign in Quartus II**, the pin mapper does not create a Quartus II pin assignment for that signal.

The **Assign in Quartus II** setting is useful for the following cases:

- You want to manually assign the pin using the Quartus II Assignment Editor.
- If a particular signal connects to logic inside the FPGA rather than to I/O pins, the signal does not need a pin assignment.

Pin Mappings Status

The pin mapper displays the current status of signal mappings. When you place the mouse over a row in the pin mapper, a tooltip displays the FPGA pin assignment that will result from the current pin mapping.

In addition, the colors of symbols in the **Source Signals** column change to indicate status. Table 8–2 on page 8–8 shows the color coding for signal mapping status.

Table 8–2. Color Coding for Signal Status

Color	Meaning	Example Cases
Red	Incomplete mapping	An individual signal is not yet mapped, or a group of signals contains one or more signals that are not yet mapped
Green	All signals mapped	The element is mapped, or the element is set to **Assign in Quartus II**.
Blue	Pins auto-assigned – This component provides its own pin assignments which cannot be overridden by the pin mapper.	The Altera DDR/DDR2 SDRAM MegaCore provides specific pin assignments which must be used with the core.
Yellow	Assign in the Quartus II software – The pin mapper cannot assign pins for this component, and therefore you must manually assign pins using the Quartus II Assignment Editor.	The pin mapper does not support components created before Quartus II version 5.0.

Section II. Building Systems with SOPC Builder

Section II of this volume 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**. Design file hyperlinks are located with individual chapters linked from the Altera web site.

This section includes the following chapters:

Revision History

The following table shows the revision history for Chapters 9 through 11.

Chapter(s)	Date / Version	Changes Made
9	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.
10	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.
11	May 2006, v6.0.0	Chapter 11 was previously chapter 10. No change to content.
	October 2005, v5.1.0	Chapter 10 was previously chapter 8. No change to content.
	May 2005, v5.0.0	No change from previous release.
	February 2005, v1.0	Initial release.

9. Building Memory Subsystems Using SOPC Builder

QII54006-6.0.0

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 the process for building a memory subsystem as part of a larger system created with SOPC Builder. This chapter focuses on the kinds of memory most commonly used in SOPC Builder systems:

- On-chip RAM and ROM
- EPCS serial configuration devices
- SDRAM
- Off-chip RAM and ROM, such as SRAM and common flash interface (CFI) flash memory

This chapter assumes that you are familiar with the following:

- Creating FPGA designs and making pin assignments with the Quartus® II software. For details, see the *Introduction to Quartus II Manual*.
- Building simple systems with SOPC Builder. For details, see the *Introduction to SOPC Builder* and *Tour of the SOPC Builder User Interface chapters* in volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, see the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon® interface. You do not need extensive knowledge of the Avalon interface, such as transfer types or signal timing. However, to create your own custom memory subsystem with external memories, you need to understand the Avalon interface. For details, see the *Avalon Switch Fabric* chapter in volume 4 of the *Quartus II Handbook* and the *Avalon Interface Specification*.

Example Design

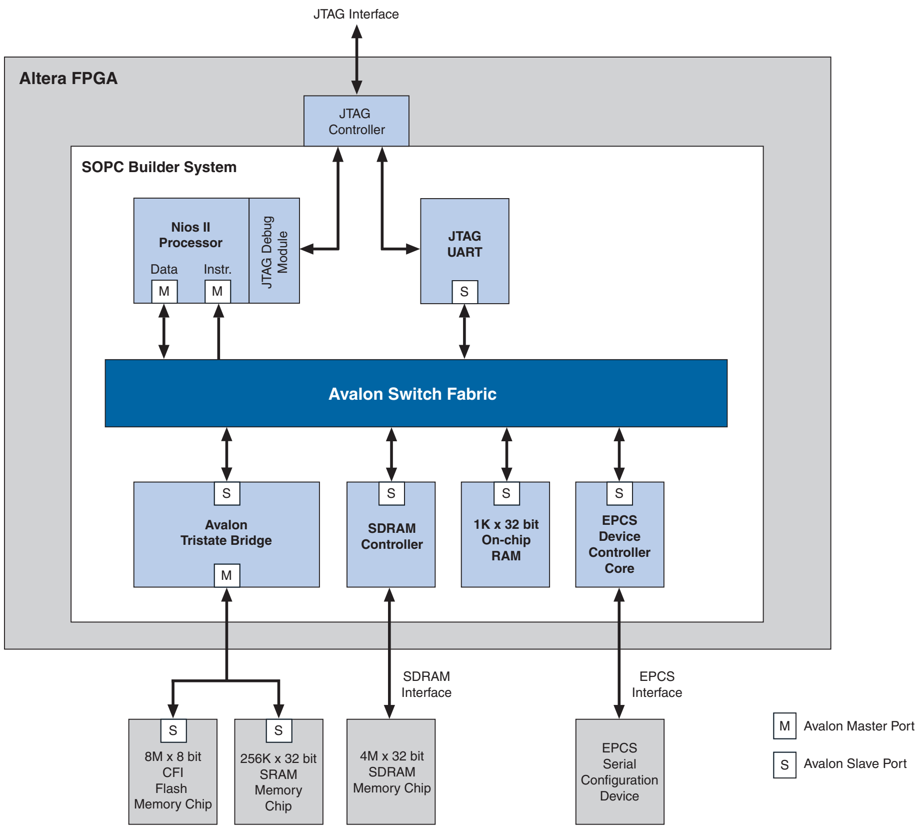
This chapter demonstrates the process for building a system that contains one of each type memory as shown in Figure 9–1. Each section of the chapter builds on previous sections, culminating in a complete system.

By following the example design through this chapter, you will learn how to create a complete memory subsystem for your own custom system. 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 don't 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 9–1 shows a block diagram of the example system.

Figure 9–1. Example Design Block Diagram



In Figure 9–1, all blocks shown below the Avalon switch 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.

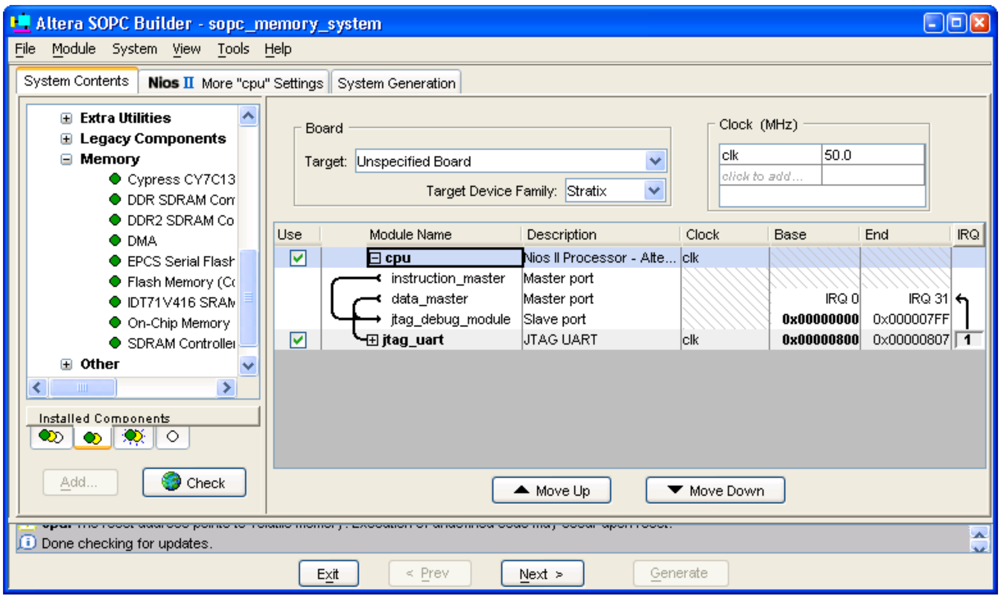
Example Design Starting Point

The following elements comprise the example design:

- A Quartus II project named **quartus2_project**.A block diagram 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 **quartus2_project** already exists, that **sopc_memory_system** has been started in SOPC Builder, and that 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 9–2 shows the starting point in SOPC Builder.

Figure 9–2. Starting Point for the Example Design



Board

Target: Unspecified Board

Target Device Family: Stratix

Clock (MHz)

clk	50.0
click to add...	

Use	Module Name	Description	Clock	Base	End	IRQ
✓	**cpu**	Nios II Processor - Alte...	clk			
	instruction_master	Master port				
	data_master	Master port		IRQ 0	IRQ 31	
	jtag_debug_module	Slave port		**0x00000000**	0x000007FF	
✓	**jtag_uart**	JTAG UART	clk	**0x00000800**	0x00000807	1

Done checking for updates.

All sections in this chapter build on this starting point.

Hardware & 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**.

This chapter does not go as far as 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 9–3:

1. Component-level design in SOPC Builder

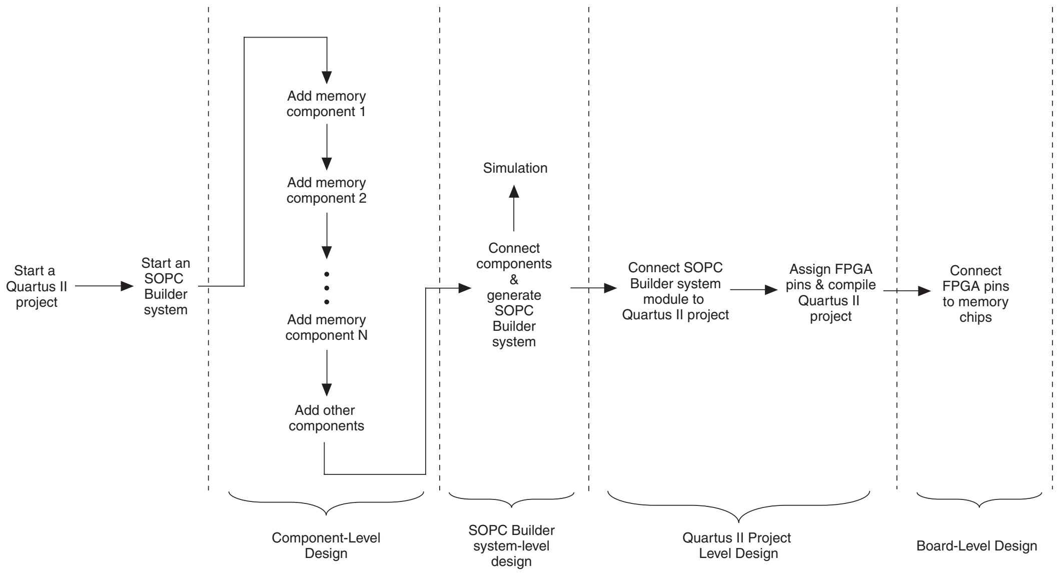
2. SOPC Builder system-level design

3. Simulation

4. Quartus II project-level design

5. Board-level design

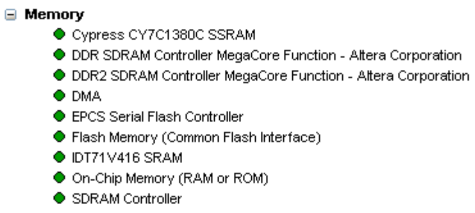
Figure 9–3. Design Flow


Start a Quartus II project
Start an SOPC Builder system
Add memory component 1
Add memory component 2
Add memory component N
Add other components
Simulation
Connect components & generate SOPC Builder system
Connect SOPC Builder system module to Quartus II project
Assign FPGA pins & compile Quartus II project
Connect FPGA pins to memory chips
Component-Level Design
SOPC Builder system-level design
Quartus II Project Level Design
Board-Level Design

Component-Level Design in SOPC Builder

In this step, you specify which memory components to use, and you configure each component to meet the needs of the system. All memory components are available from the **Memory** category in the SOPC Builder list of available components, shown in Figure 9–4.

Figure 9–4. List of Available Memory Components in SOPC Builder



Memory
- Cypress CY7C1380C SSRAM
- DDR SDRAM Controller MegaCore Function - Altera Corporation
- DDR2 SDRAM Controller MegaCore Function - Altera Corporation
- DMA
- EPCS Serial Flash Controller
- Flash Memory (Common Flash Interface)
- IDT71V416 SRAM
- On-Chip Memory (RAM or ROM)
- SDRAM Controller

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 for 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 testbench generated by SOPC Builder. See "Simulation Considerations" on page 9–7.

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, then you must make board-level design choices.

Simulation Considerations

SOPC Builder can automatically generate a testbench for register transfer level (RTL) simulation of the system. This testbench instantiates the system module and can also instantiate memory models for external memory components. The testbench is plain-text hardware description language (HDL), located at the bottom of the top-level system module HDL design file. To explore the contents of the auto-generated testbench, open the top-level HDL file, and search on keyword `test_bench`.

Generic Memory Models

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

The generic memory models store memory initialization files, such as Data [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 users, 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.

CAUTION There are special sections of the system module design file that you can edit safely. You must edit only these sections, because SOPC Builder overwrites the rest of the system module every time you generate the system. These sections are marked by the following text:

Verilog HDL

```
// <ALTERA_NOTE> CODE INSERTED BETWEEN HERE
//  add your signals and additional architecture here
// AND HERE WILL BE PRESERVED </ALTERA_NOTE>
```

VHDL

```
-- <ALTERA_NOTE> CODE INSERTED BETWEEN HERE
--add your component and signal declaration here
-- AND HERE WILL BE PRESERVED </ALTERA_NOTE>
```

On-Chip RAM & 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 adding an **On-chip Memory (RAM or ROM)** 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 (writeable)** – This setting creates a readable and writeable 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 forces 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:
  - **Automatic** – This setting allows the Quartus II software to choose the most appropriate memory resource.
  - **M512** – This setting forces the Quartus II software to use M512 blocks.
  - **M4K** – This setting forces the Quartus II software to use M4K blocks.
  - **M-RAM** – This setting forces 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.

- **Memory Width** – This setting determines the data width of the memory. The available choices are 8, 16, 32, 64, or 128 bits. Assign **Memory 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 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.

SOPC Builder System-Level Design for On-Chip Memory

There are not many SOPC Builder system-level design considerations for on-chip memories. See "SOPC Builder System-Level Design" on page 9–7.

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 memory initialization file automatically.

Board-Level Design for On-Chip Memory

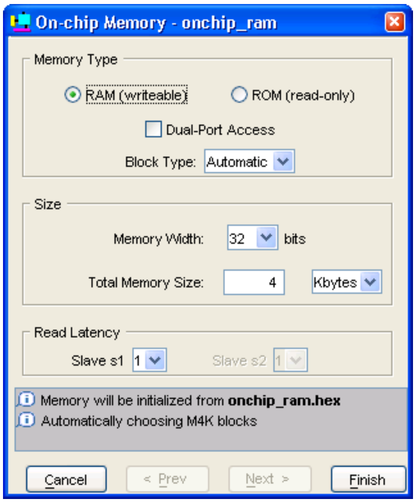
The on-chip memory is embedded inside the SOPC Builder system module, and therefore there is nothing to connect at the board level.

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 9–5 shows the **On-Chip Memory (RAM or ROM)** configuration wizard settings for the example design.

Figure 9–5. On-Chip Memory (RAM or ROM) Configuration Wizard



Memory Type

[✓ RAM (writeable)] ROM (read-only)

Dual-Port Access

Block Type: Automatic

Size

Memory Width: 32 bits

Total Memory Size: 4 Kbytes

Read Latency

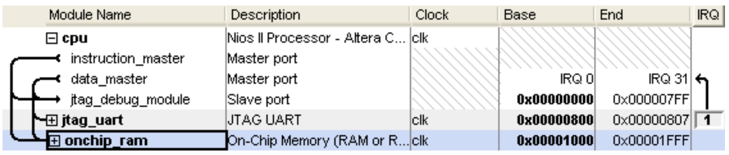
Slave s1 1 Slave s2 1

Memory will be initialized from **onchip_ram.hex**

Automatically choosing M4K blocks

Figure 9–6 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.

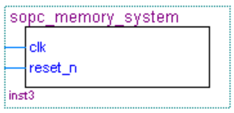
Figure 9–6. SOPC Builder System with On-Chip Memory



Module Name	Description	Clock	Base	End	IRQ
cpu	Nios II Processor - Altera C...	clk			
instruction_master	Master port				
data_master	Master port		IRQ 0	IRQ 31	
jtag_debug_module	Slave port		**0x00000000**	0x000007FF	
jtag_uart	JTAG UART	clk	**0x00000800**	0x00000807	1
onchip_ram	On-Chip Memory (RAM or R...	clk	**0x00001000**	0x00001FFF	

For demonstration purposes, Figure 9–7 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 9–7. 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 on the features and usage of the EPCS device controller core, see 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 is only one setting for this component: **Reference Designator**. When targeting a board that declares a reference designator for the EPCS device, the **Reference Designator** setting is fixed.

SOPC Builder uses reference designators to specify a unique identifier for flash memory devices on the board. This convention is a requirement of the Nios II EDS, specifically the Nios II Flash Programmer utility.

For details, see 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 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

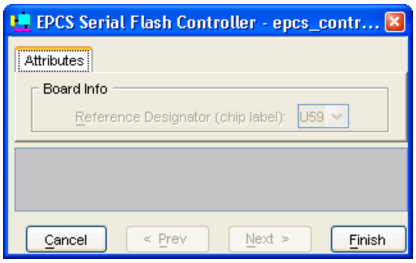
You must connect the EPCS device to the FPGA as described in the Altera *Configuration Handbook*. No other connections are necessary.

Example Design with an EPCS Device

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

Figure 9–8 shows the **EPCS Serial Flash Controller** configuration wizard settings for the example design. In this example, the target board declares a reference designator U59 for the EPCS device on the board.

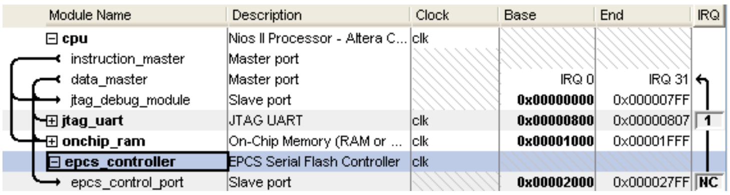
Figure 9–8. EPCS Serial Flash Controller Configuration Wizard



Reference Designator (chip label): U59

Figure 9–9 shows the SOPC Builder system after adding an instance of the EPCS controller core and assigning it a base address.

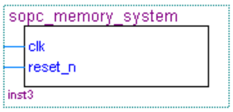
Figure 9–9. SOPC Builder System with EPCS Device



Module Name	Description	Clock	Base	End	IRQ
cpu	Nios II Processor - Altera C...	clk			
instruction_master	Master port				
data_master	Master port		IRQ 0	IRQ 31	
jtag_debug_module	Slave port		**0x00000000**	0x000007FF	
jtag_uart	JTAG UART	clk	**0x00000800**	0x00000807	1
onchip_ram	On-Chip Memory (RAM or ...	clk	**0x00001000**	0x00001FFF	
epcs_controller	EPCS Serial Flash Controller	clk			
epcs_control_port	Slave port		**0x00002000**	0x000027FF	NC

For demonstration purposes only, Figure 9–10 shows the result of generating the system module at this stage.

Figure 9–10. 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, see Altera's *Configuration Handbook*.

SDRAM

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

For further details on the features and usage of the SDRAM controller core, see 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, etc.) and the timing specifications of the device(s) on the board.

For complete details on configuration options for the SDRAM controller core, see the *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

SOPC Builder System-Level Design for SDRAM

In SOPC Builder on the System Contents tab, the SDRAM controller looks like any other memory component. Similar to on-chip memory, there are not many SOPC Builder system-level design considerations for SDRAM. See "SOPC Builder System-Level Design" on page 9–7.

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, see "Simulation Considerations" on page 9–7 and "Hardware Simulation Considerations" in the chapter *SDRAM Controller Core with Avalon Interface* 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 & 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 overcome clock-skew issues. The exact settings for the `altpll` depend on your target hardware; you must experiment to tune the phase shift to match the board.

For details, see the *altpll Megafunction User Guide*.

Board-Level Design for SDRAM

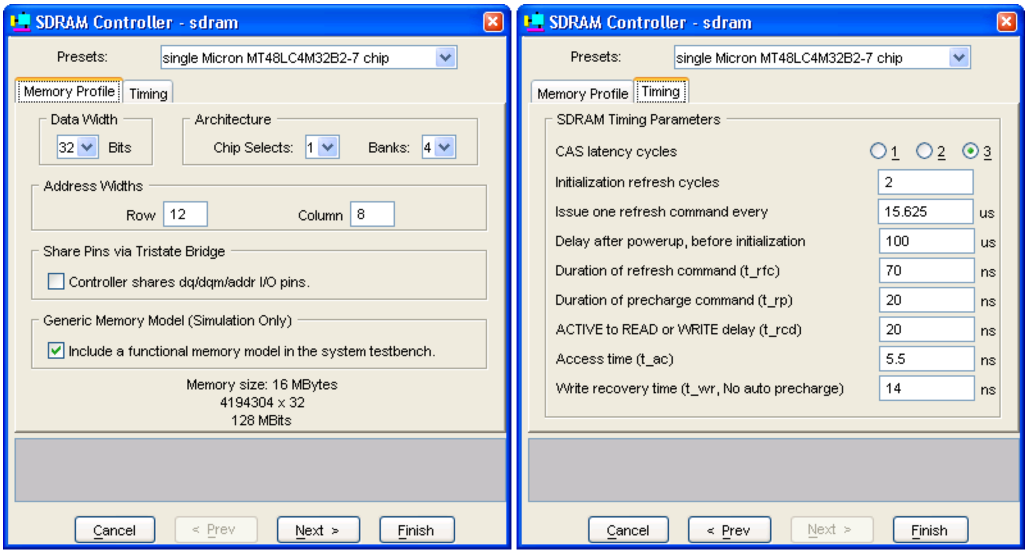
Memory requirements largely dictate the board-level configuration of the SDRAM device(s). The SDRAM controller core can accommodate various configurations of SDRAM on the board, including multiple banks and multiple devices.

For further details, see the "Example Configurations" section in 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 9–11 shows the **SDRAM Controller** configuration wizard settings for the example design.

Figure 9–11. SDRAM Controller Configuration Wizard


SDRAM Controller - sdram

Presets: single Micron MT48LC4M32B2-7 chip

Memory Profile

Data Width: 32 Bits

Architecture: Chip Selects: 1 Banks: 4

Address Widths: Row 12 Column 8

Share Pins via Tristate Bridge

☐ Controller shares dq/dqm/addr I/O pins.

Generic Memory Model (Simulation Only)

☑ Include a functional memory model in the system testbench.

Memory size: 16 MBytes
4194304 x 32
128 MBits

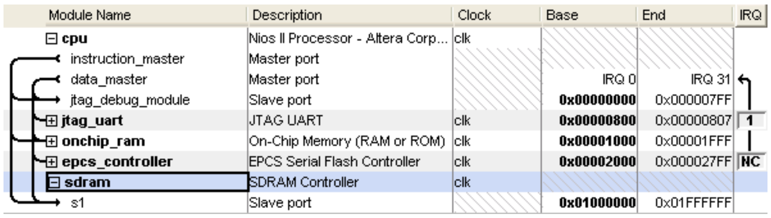
Timing

SDRAM Timing Parameters

CAS latency cycles	1 2 [✓ 3]	
Initialization refresh cycles	2	
Issue one refresh command every	15.625	us
Delay after powerup, before initialization	100	us
Duration of refresh command (t_rfc)	70	ns
Duration of precharge command (t_rp)	20	ns
ACTIVE to READ or WRITE delay (t_rcd)	20	ns
Access time (t_ac)	5.5	ns
Write recovery time (t_wr, No auto precharge)	14	ns

Figure 9–12 shows the SOPC Builder system after adding an instance of the SDRAM controller, renaming it to **sdram**, and assigning it a base address.

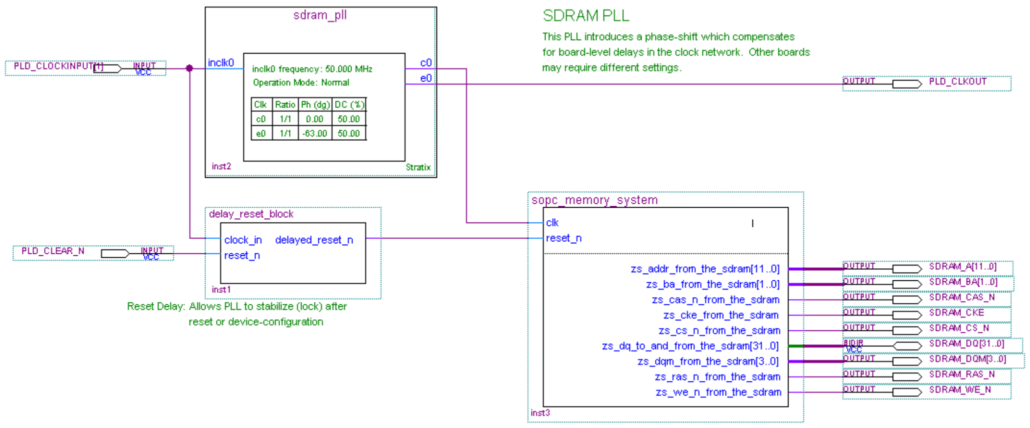
Figure 9–12. SOPC Builder System with SDRAM



Module Name	Description	Clock	Base	End	IRQ
⊟ **cpu**	Nios II Processor - Altera Corp...	clk			
instruction_master	Master port				
data_master	Master port		IRQ 0	IRQ 31	
jtag_debug_module	Slave port		**0x00000000**	0x000007FF	
⊞ **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			
s1	Slave port		**0x01000000**	0x01FFFFFF	

For demonstration purposes, Figure 9–13 shows the result of generating the system module at this stage, and connecting it in **toplevel_design.bdf**.

Figure 9–13. toplevel_design.bdf with SDRAM



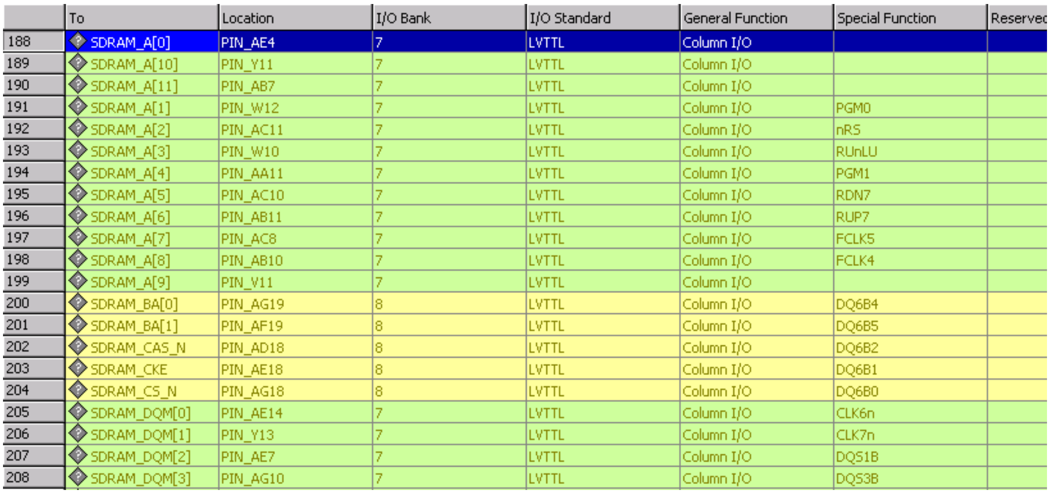
After generating the system, the top-level system module file **sopc_memory_system.v** contains the list of SDRAM-related I/O signals which must be connected to FPGA pins:

```
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 9–13, **toplevel_design.bdf** uses an instance of `sdram_pll` to phase shift the SDRAM clock by -63 degrees. **toplevel_design.bdf** also uses a subdesign `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 9–14 shows pin assignments in the Quartus II assignment editor for some of the SDRAM pins. The correct pin assignments depend on the target board.

Figure 9–14. Pin Assignments for SDRAM



	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reservec
188	SDRAM_A[0]	PIN_AE4	7	LVTTL	Column I/O		
189	SDRAM_A[10]	PIN_Y11	7	LVTTL	Column I/O		
190	SDRAM_A[11]	PIN_AB7	7	LVTTL	Column I/O		
191	SDRAM_A[1]	PIN_W12	7	LVTTL	Column I/O	PGM0	
192	SDRAM_A[2]	PIN_AC11	7	LVTTL	Column I/O	nRS	
193	SDRAM_A[3]	PIN_W10	7	LVTTL	Column I/O	RUnLU	
194	SDRAM_A[4]	PIN_AA11	7	LVTTL	Column I/O	PGM1	
195	SDRAM_A[5]	PIN_AC10	7	LVTTL	Column I/O	RDN7	
196	SDRAM_A[6]	PIN_AB11	7	LVTTL	Column I/O	RUP7	
197	SDRAM_A[7]	PIN_AC8	7	LVTTL	Column I/O	FCLK5	
198	SDRAM_A[8]	PIN_AB10	7	LVTTL	Column I/O	FCLK4	
199	SDRAM_A[9]	PIN_V11	7	LVTTL	Column I/O		
200	SDRAM_BA[0]	PIN_AG19	8	LVTTL	Column I/O	DQ6B4	
201	SDRAM_BA[1]	PIN_AF19	8	LVTTL	Column I/O	DQ6B5	
202	SDRAM_CAS_N	PIN_AD18	8	LVTTL	Column I/O	DQ6B2	
203	SDRAM_CKE	PIN_AE18	8	LVTTL	Column I/O	DQ6B1	
204	SDRAM_CS_N	PIN_AG18	8	LVTTL	Column I/O	DQ6B0	
205	SDRAM_DQM[0]	PIN_AE14	7	LVTTL	Column I/O	CLK6n	
206	SDRAM_DQM[1]	PIN_Y13	7	LVTTL	Column I/O	CLK7n	
207	SDRAM_DQM[2]	PIN_AE7	7	LVTTL	Column I/O	DQS1B	
208	SDRAM_DQM[3]	PIN_AG10	7	LVTTL	Column I/O	DQS3B	

Off-Chip SRAM & 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 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 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 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 & 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 development boards.

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 **AvalonTristateBridge** component as well. Multiple off-chip memories can connect to a single tristate bridge.

Avalon Tristate Bridge

A tristate bridge connects off-chip devices to on-chip Avalon switch 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 Avalon switch 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 **AvalonTristateBridge** component. The Avalon 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 Avalon switch fabric.

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

Registering the input and output signals shortens the register-to-register delay from the memory device to the FPGA, resulting in higher system f_{MAX} performance. However, in each direction, the registers add one additional cycle of latency for Avalon 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 on the Avalon tristate interface, refer to the *Avalon 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 on features and usage, refer to chapter *Common Flash Interface Controller Core with Avalon Interface* 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 & Flash Memory" on page 9–26.

SRAM

To instantiate an interface to off-chip RAM, you 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 on 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 & Flash Memory

In SOPC Builder on the System Contents tab, the Avalon tristate bridge has two ports:

- Avalon 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 tristate master port – This port faces the off-chip memory devices. You connect this master port to the Avalon tristate slave ports on the interface components for off-chip memories.

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

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

Simulation for SRAM & 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 testbench. 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 testbench.

For further details, see "Simulation Considerations" on page 9–7.

Quartus II Project-Level Design for SRAM & 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. An example is below:

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

Board-Level Design for SRAM & Flash Memory

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

Special consideration is required when connecting the Avalon `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 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 `address` lines. For example, a 16-bit memory device must ignore Avalon `address[0]` (which is a byte address), and connect Avalon `address[1]` to the least-significant address line.

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

Table 9–1. Connecting the Least-Significant Avalon Address Line

Avalon 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 `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, then the smaller memory ignores the ten most-significant Avalon address lines.

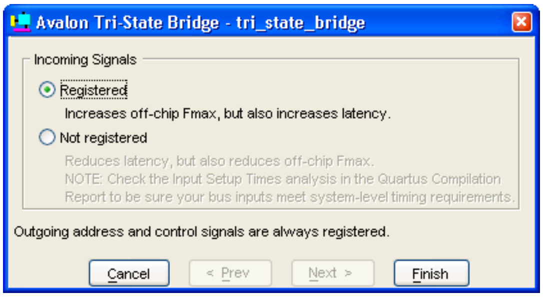
Example Design with SRAM & Flash Memory

This section demonstrates adding a 1 Mbyte SRAM and an 8Mbyte flash memory to the example design. These memory devices connect to the Avalon switch fabric through an Avalon tristate bridge.

Adding the Avalon Tristate Bridge

Figure 9–15 shows the **Avalon Tristate Bridge** configuration wizard for the example design. The example design uses registered inputs and outputs to maximize system f_{MAX}, which increases the read latency by two for both the SRAM and flash memory.

Figure 9–15. Avalon Tristate Bridge Configuration Wizard


Avalon Tri-State Bridge - tri_state_bridge

Incoming Signals

[✓ Registered]

Increases off-chip Fmax, but also increases latency.

Not registered

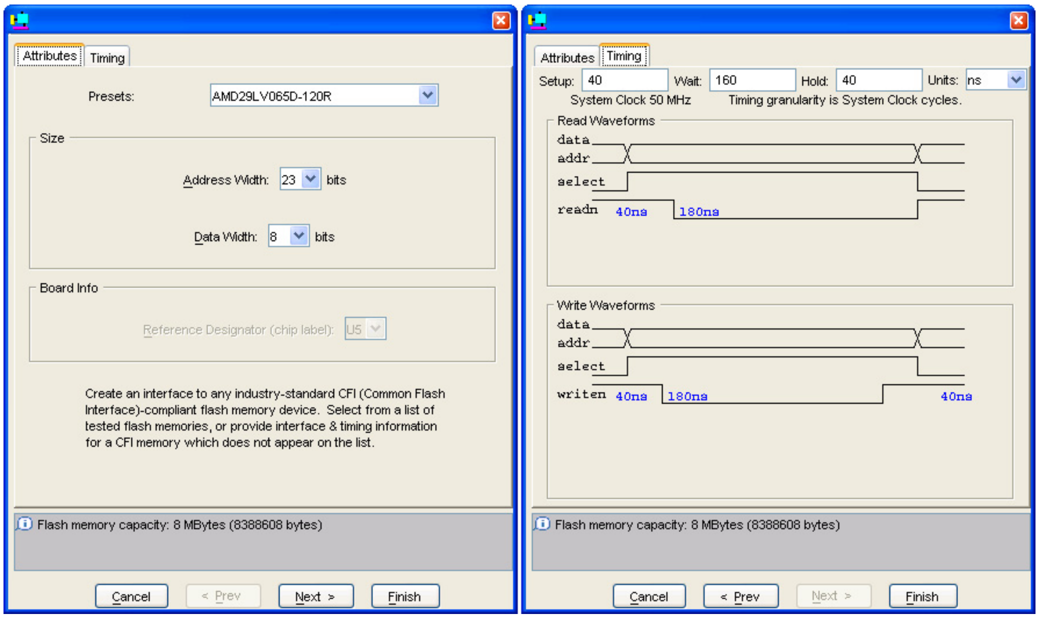
Reduces latency, but also reduces off-chip Fmax.
NOTE: Check the Input Setup Times analysis in the Quartus Compilation Report to be sure your bus inputs meet system-level timing requirements.

Outgoing address and control signals are always registered.

Adding the Flash Memory Interface

The flash memory is 8M x 8-bit, which requires 23 address bits and 8 data bits. Figure 9–16 shows the **Flash Memory (Common Flash Interface)** configuration wizard settings for the example design. In this example, the target board declares a reference designator U5 for the flash device on the board.

Figure 9–16. Flash Memory Configuration Wizard



Presets: AMD29LV065D-120R

Size

Address Width: 23 bits

Data Width: 8 bits

Board Info

Reference Designator (chip label): U5

Create an interface to any industry-standard CFI (Common Flash Interface)-compliant flash memory device. Select from a list of tested flash memories, or provide interface & timing information for a CFI memory which does not appear on the list.

Flash memory capacity: 8 MBytes (8388608 bytes)

Setup: 40 Wait: 160 Hold: 40 Units: ns

System Clock 50 MHz Timing granularity is System Clock cycles.

Flash memory capacity: 8 MBytes (8388608 bytes)

Adding the SRAM Interface

The SRAM device is 256K x 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. Figure 9–17 – Figure 9–21 shows the settings required on the various component editor tabs to implement an interface to this SRAM.

Figure 9–17. SRAM Interface Component Editor HDL Files Tab

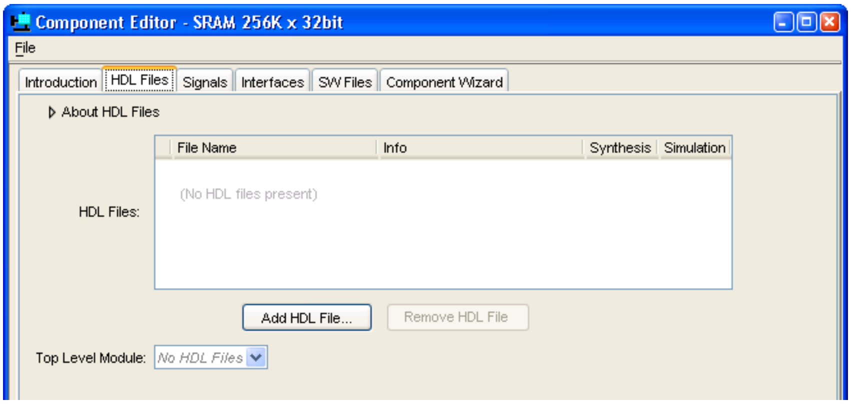
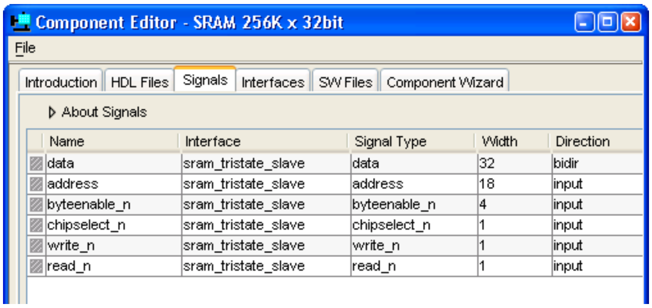
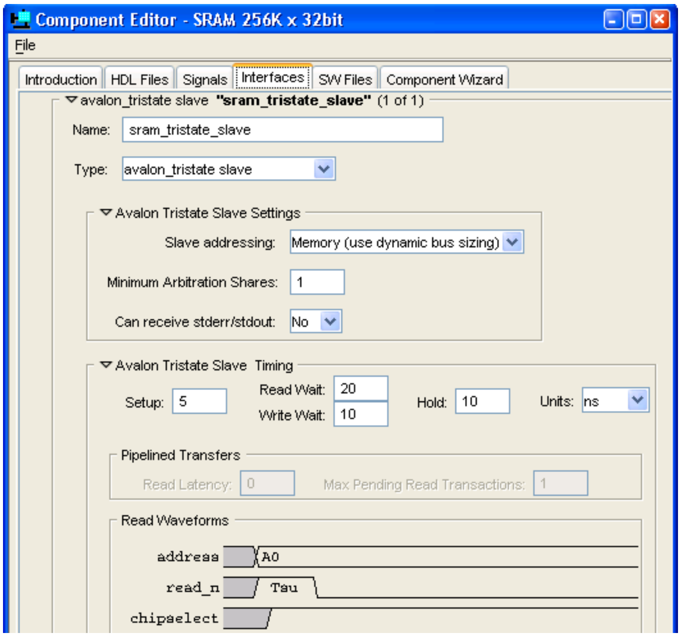


Figure 9–18. SRAM Interface Component Editor Signals Tab



Name	Interface	Signal Type	Width	Direction
data	sram_tristate_slave	data	32	bidir
address	sram_tristate_slave	address	18	input
byteenable_n	sram_tristate_slave	byteenable_n	4	input
chipselect_n	sram_tristate_slave	chipselect_n	1	input
write_n	sram_tristate_slave	write_n	1	input
read_n	sram_tristate_slave	read_n	1	input

Figure 9–19. SRAM Interface Component Editor Interfaces Tab



avalon_tristate slave **"sram_tristate_slave"** (1 of 1)

Name: sram_tristate_slave

Type: avalon_tristate slave

Avalon Tristate Slave Settings

Slave addressing: Memory (use dynamic bus sizing)

Minimum Arbitration Shares: 1

Can receive stderr/stdout: No

Avalon Tristate Slave Timing

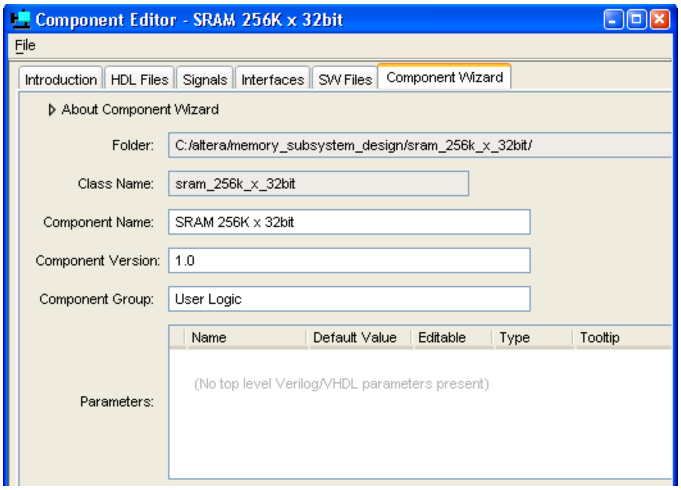
Setup: 5 Read Wait: 20 Write Wait: 10 Hold: 10 Units: ns

Pipelined Transfers

Read Latency: 0 Max Pending Read Transactions: 1

Read Waveforms

Figure 9–20. SRAM Interface Component Editor Component Wizard Tab



Folder: C:/altera/memory_subsystem_design/sram_256k_x_32bit/

Class Name: sram_256k_x_32bit

Component Name: SRAM 256K x 32bit

Component Version: 1.0

Component Group: User Logic

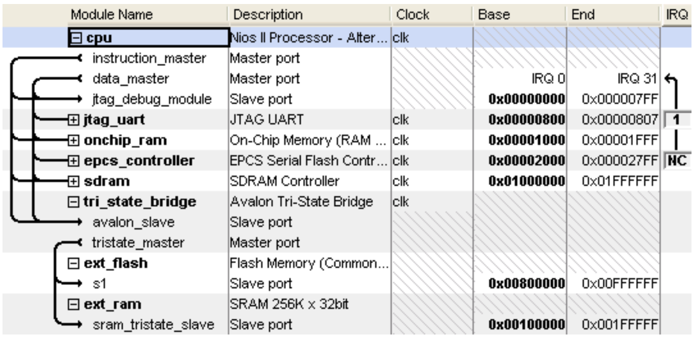
Parameters:

Name	Default Value	Editable	Type	Tooltip
(No top level Verilog/VHDL parameters present)				

SOPC Builder System Contents Tab

Figure 9–21 shows the SOPC Builder system after adding the tristate bridge and memory interface components, and configuring them appropriately on the **System Contents** tab. Figure 9–21 represents the complete example design in SOPC Builder.

Figure 9–21. SOPC Builder System with SRAM & Flash Memory



Module Name	Description	Clock	Base	End	IRQ
cpu	Nios II Processor - Alter...	clk			
instruction_master	Master port				
data_master	Master port		IRQ 0	IRQ 31	
jtag_debug_module	Slave port		**0x00000000**	0x000007FF	
jtag_uart	JTAG UART	clk	**0x00000800**	0x00000807	1
onchip_ram	On-Chip Memory (RAM ...	clk	**0x00001000**	0x00001FFF	
epcs_controller	EPCS Serial Flash Contr...	clk	**0x00002000**	0x000027FF	NC
sdram	SDRAM Controller	clk	**0x01000000**	0x01FFFFFF	
tri_state_bridge	Avalon Tri-State Bridge	clk			
avalon_slave	Slave port				
tristate_master	Master port				
ext_flash	Flash Memory (Common...				
s1	Slave port		**0x00800000**	0x00FFFFFF	
ext_ram	SRAM 256K x 32bit				
sram_tristate_slave	Slave port		**0x00100000**	0x001FFFFF	

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 which 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_byteenablen;
inout    [ 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 tristate bridge signals which can be shared are named after the instance of the tristate bridge component, such as `tri_state_bridge_data[31:0]`.

Connecting & Assigning Pins in the Quartus II Project

Figure 9–22 shows the result of generating the system module for the complete example design, and connecting it in **toplevel_design.bdf**.

Figure 9–22. toplevel_design.bdf with SRAM & Flash Memory

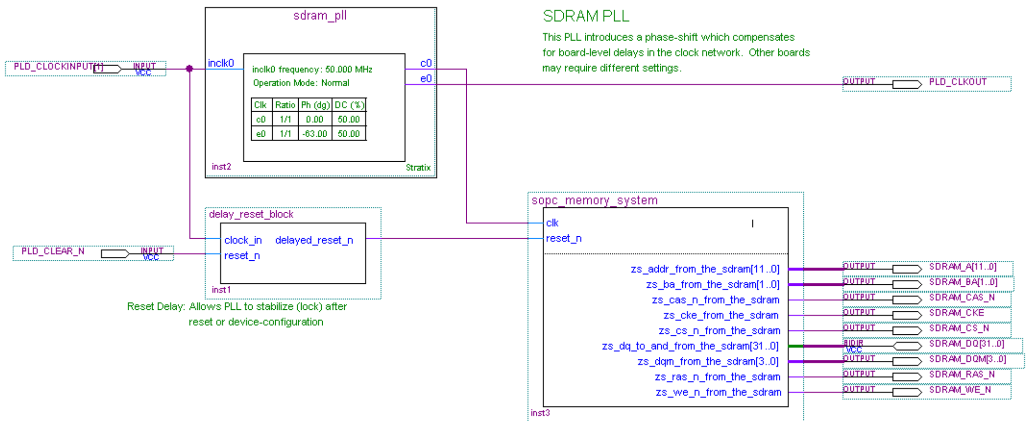
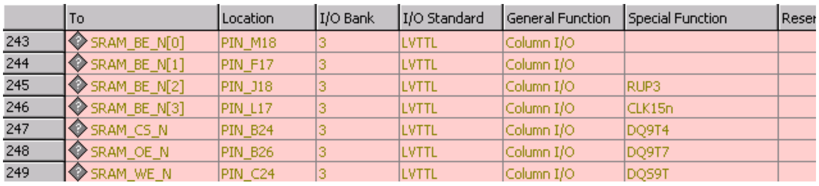


Figure 9–23 shows the pin assignments in the Quartus II assignment editor for some of the SRAM and flash memory pins. The correct pin assignments depend on the target board.

Figure 9–23. Pin Assignments for SRAM & Flash Memory



	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reser
243	SRAM_BE_N[0]	PIN_M18	3	LVTTL	Column I/O		
244	SRAM_BE_N[1]	PIN_F17	3	LVTTL	Column I/O		
245	SRAM_BE_N[2]	PIN_J18	3	LVTTL	Column I/O	RUP3	
246	SRAM_BE_N[3]	PIN_L17	3	LVTTL	Column I/O	CLK15n	
247	SRAM_CS_N	PIN_B24	3	LVTTL	Column I/O	DQ9T4	
248	SRAM_OE_N	PIN_B26	3	LVTTL	Column I/O	DQ9T7	
249	SRAM_WE_N	PIN_C24	3	LVTTL	Column I/O	DQS9T	

Connecting FPGA Pins to Devices on the Board

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

Table 9–2. FPGA Connections to SRAM & Flash Memory (Part 1 of 2)

Avalon Address Line	Flash Address (8M x 8-bit Data)	SRAM Address (256K x 32-bit data)
`tri_state_bridge_address[0]`	A0	No connect
`tri_state_bridge_address[1]`	A1	No connect
`tri_state_bridge_address[2]`	A2	A0

Table 9–2. FPGA Connections to SRAM & Flash Memory (Part 2 of 2)

Avalon Address Line	Flash Address (8M x 8-bit Data)	SRAM Address (256K x 32-bit data)
`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	A14
`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

10. Developing Components for SOPC Builder

QII54007-6.0.0

Introduction

This chapter describes the design flow to develop a custom SOPC Builder component. This chapter provides tutorial steps that guide you through the process of creating a custom component, integrating it into a system, and downloading it to hardware.

This chapter is divided into the following sections:

- *Component Development Flow* (see page 10–3).
- *Design Example: Pulse-Width Modulator (PWM) Slave* (see page 10–8). This design example demonstrates developing a component with a single Avalon® slave interface. In this section, you will start with a ready-made HDL design, package it into a SOPC Builder component, and then instantiate it in a system. If you have a development board, you can download the design to hardware and see the PWM work.
- *Sharing Components* (see page 10–28). This section shows you how to relocate component files to use them in other systems, or share them with other designers.

SOPC Builder Components & the Component Editor

SOPC Builder provides a component editor that lets you create and edit your own SOPC Builder components. By following the procedures described in this document, you will learn to use the component editor and turn any custom logic module into an SOPC Builder component.

Once your custom logic is packaged as component, you can instantiate it in an SOPC Builder system in the same manner as commercially available SOPC Builder Ready components. You can share your component with other designers to encourage design reuse.

Typically, a component is comprised of the following:

- Hardware files: HDL modules that describe the component hardware
- Software files: A C-language header file that defines the component register map, and driver software that allows programs to control the component
- Component description file (**class.ptf**): This file defines the structure of the component, and provides SOPC Builder the information it needs to integrate the component into a system. The component

editor generates this file automatically based on the hardware & software files you provide, and the parameters you specify in the component editor GUI.

After you create the hardware and software files that describe the component, you use the component editor to package those file into an SOPC Builder component. You can also use the component editor later to re-edit the component, if you ever update the hardware or software files.

Assumptions About the Reader

This chapter assumes that you are familiar with the following:

- Building systems with SOPC Builder. For details, see the *Introduction to SOPC Builder* and *Tour of the SOPC Builder User Interface* chapters in volume 4 of the *Quartus II Handbook.*
- SOPC Builder components. For details, see the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon interface. You do not need extensive knowledge of the Avalon interface, such as transfer types or signal timing, to use the design example(s) provided with this chapter. However, to create your own custom components, you need a fuller understanding of the Avalon interface. For details, see the *Avalon Interface Specification*.

Hardware & Software Requirements

To use the design example(s) in this chapter, you must have the following:

- Design files for the example design – A hyperlink to the design files appears next to this chapter on the SOPC Builder literature page. Visit **www.altera.com/sopcbuilder**.
- Quartus® II Software version 4.2 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® II Edition
  - Stratix Edition
  - Stratix Professional Edition
  - Cyclone™ Edition

If you do not have a development board, you can follow the hardware development steps, but you will not be able to 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**.

Before you begin, you must install the Quartus® II software and Nios II development tools.

Component Development Flow

This section provides an overview of the development process for SOPC Builder components, covering both the hardware and software aspects. This section focuses on the design flow for components with a single Avalon slave interface. However, these steps are easily extrapolated to components with a master port, or multiple master and slave ports.

Typical Design Steps

A typical development sequence for a slave component includes the following steps, not necessarily in this order:

1. Specify the hardware functionality.
2. If a microprocessor will be used to control the component, specify the application program interface (API) to access and control the hardware.
3. Based on the hardware and software requirements, define an Avalon interface that provides:
   a. Appropriate control mechanisms
   b. Adequate throughput performance
4. Write HDL that describes the hardware in either Verilog or VHDL.
5. Test the component hardware alone to verify correct operation.
6. Write a C header file that defines the hardware-level register map for software.
7. Use the component editor to package the initial hardware and software files into a component.

8. Instantiate the component into a simple SOPC Builder system module.

9. Test register-level accesses to the component using a microprocessor, such as the Nios II processor. You can perform verification in hardware, or on an HDL simulator such as ModelSim®.

10. If a microprocessor will be used to control the component, write driver software.

11. Iteratively improve the component design, based on in-system behavior of the component:

    a. Make hardware improvements and adjustments.

    b. Make software improvements and adjustments.

    c. Incorporate hardware and software changes into the component using the component editor.

12. Build a complete SOPC Builder system incorporating one or more instances of the component.

13. Perform system-level verification. Make further iterative improvements, if necessary.

14. Finalize the component and distribute it for design reuse.

The design process for a master component is similar, except for software development aspects.

Hardware Design

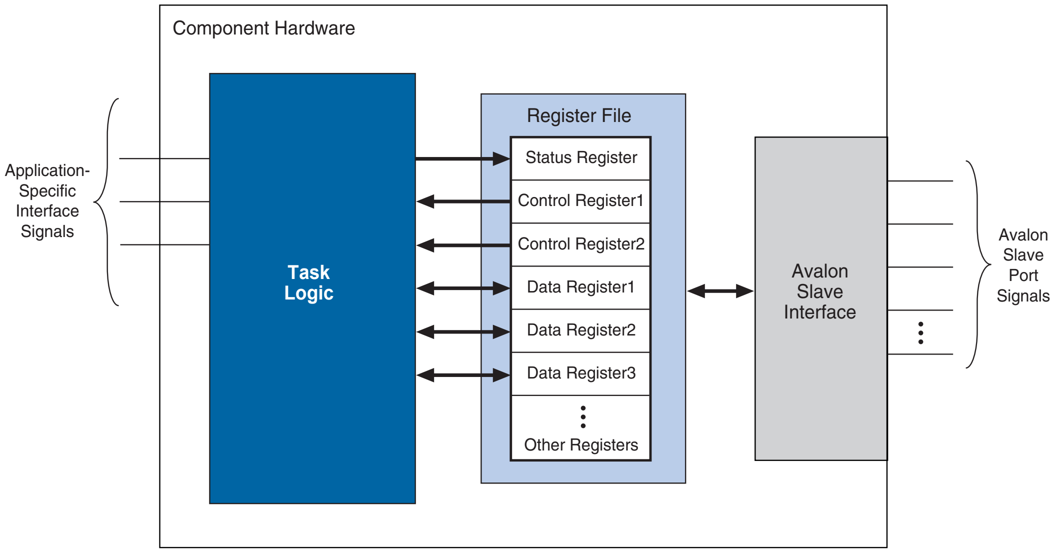
As with any logic design process, the development of SOPC Builder component hardware begins after the specification phase. Coding the HDL is an iterative process, as you write and verify the HDL logic against the specification.

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

- *Task Logic* - The task logic implements the component's fundamental function. The task logic is design dependent.
- *Register File* - The register file provides a path for communicating signals from inside the task logic to the outside world, and vice versa. The register file maps internal nodes to addressable offsets that can be read or written by the Avalon interface.
- *Avalon Interface* - The Avalon interface provides a standard Avalon front-end to the register file. The interface uses any Avalon signal types necessary to access the register file and support the transfer types required by the task logic. The following factors affect the Avalon interface:
  - How wide is the data to be transferred?
  - What is the throughput requirement for the data transfers?
  - Is this interface primarily for control or for data? That is, do transfers tend to be sporadic, or come in continuous bursts?
  - Is the hardware relatively fast or slow compared to other components that will be in a system?

Figure 10–1 shows a block diagram of a typical component with one Avalon slave port.

Figure 10–1. Typical Component with One Avalon Slave Port



Software Design

If your intent is for 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 slave port that is accessible to a processor. The component editor lets you package a C header file with the component to define the software view of the hardware.

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 an excerpt from the register map for an Altera-provided UART component for the Nios II processor.

Example: Register Map for a Component

```
#include <io.h>
#define IOADDR_ALTERA_AVALON_TIMER_STATUS(base)        __IO_CALC_ADDRESS_NATIVE(base, 0)
#define IORD_ALTERA_AVALON_TIMER_STATUS(base)          IORD(base, 0)
#define IOWR_ALTERA_AVALON_TIMER_STATUS(base, data)    IOWR(base, 0, data)

#define ALTERA_AVALON_TIMER_STATUS_TO_MSK              (0x1)
#define ALTERA_AVALON_TIMER_STATUS_TO_OFST             (0)
#define ALTERA_AVALON_TIMER_STATUS_RUN_MSK             (0x2)
#define ALTERA_AVALON_TIMER_STATUS_RUN_OFST            (1)

#define IOADDR_ALTERA_AVALON_TIMER_CONTROL(base)       __IO_CALC_ADDRESS_NATIVE(base, 1)
#define IORD_ALTERA_AVALON_TIMER_CONTROL(base)         IORD(base, 1)
#define IOWR_ALTERA_AVALON_TIMER_CONTROL(base, data)   IOWR(base, 1, data)

#define ALTERA_AVALON_TIMER_CONTROL_ITO_MSK            (0x1)
#define ALTERA_AVALON_TIMER_CONTROL_ITO_OFST           (0)
#define ALTERA_AVALON_TIMER_CONTROL_CONT_MSK           (0x2)
#define ALTERA_AVALON_TIMER_CONTROL_CONT_OFST          (1)
#define ALTERA_AVALON_TIMER_CONTROL_START_MSK          (0x4)
#define ALTERA_AVALON_TIMER_CONTROL_START_OFST         (2)
#define ALTERA_AVALON_TIMER_CONTROL_STOP_MSK           (0x8)
#define ALTERA_AVALON_TIMER_CONTROL_STOP_OFST          (3)
```

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.

Driver software is dependent on the target processor. The component editor lets you easily package software drivers for the hardware abstraction layer (HAL) used by the Nios II processor development tools. To provide drivers for other processors, you must accommodate the needs of the development tools for the target processor.

For details on writing drivers for the Nios II HAL, see the *Nios II Software Developer's Handbook*. 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 <*Nios II EDS install path*>**/components/**.

Verifying the Component

You can verify the component in incremental stages, as you complete more and more of the design. Typically, you first verify the hardware logic as a unit (which might comprise 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 behavioral or register transfer level (RTL) simulation tools. Similarly, you can verify all component logic, including the register file and the Avalon interface(s), using your preferred verification 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. The results can be verified either on the ModelSim simulator or on hardware, such as a Nios development board.

See *AN351: Simulating Nios II Embedded Processor Designs f*or more information.

System-Level Verification

After you package the HDL files into a component using 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 RTL simulators such as ModelSim. While SOPC Builder produces a testbench for system-level verification, the capability of the simulation environment is largely dependent on the components included in the system.

☞ During the verification phase, including a Nios II processor in the system can be useful to get the benefits of the Nios II simulation environment. Even if your component has no relationship to the Nios II processor, the auto-generated ModelSim simulation environment provides an easy-to-use base that you can build upon to verify other logic in the system.

Design Example: Pulse-Width Modulator Slave

This section uses a pulse-width modulator (PWM) design example to demonstrate the steps to create a component and instantiate it in a system. This component has a single Avalon slave port.

In this section, you will perform the following steps:

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

Install the Design Files

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

⚠ Do not use spaces in any directory path names when installing the design files. If the path contains spaces, SOPC Builder might not be able to access the files.

Perform the following steps to setup the design environment:

1. Unzip the contents of the PWM zip file to a directory on your computer. This document will refer to this directory as the <*PWM design files*> directory.

2. 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 version of the design. You can use either one. Table 10–1 shows the names of the directories for the available Nios development boards.

Table 10–1. Design File Directories

Nios Development Board	Tutorial Directory
Stratix II Edition	niosII_stratixII_2s60_es
Stratix Edition	niosII_stratix_1s10 or niosII_stratix_1s10_es
Stratix Professional Edition	niosII_stratix_1s40
Cyclone Edition	niosII_cyclone_1c20

   For demonstration purposes, the figures in this chapter show the case of the Verilog design on the Nios Development Board, Cyclone Edition.

3. Copy the **standard** directory to a new location. By copying the design files, you avoid corrupting the original design. This document will refer to the newly-created directory as the **<*Quartus II project*>** directory.

Review the Example Design Specifications

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

- PWM Design Files
- Functional Specification
- PWM Task Logic
- Register File
- Avalon Interface
- Software API

In a typical design flow, it is the designer's responsibility to specify the behavior of the component.

PWM Design Files

Table 10–2 lists the contents provided in the <*PWM design files*> directory.

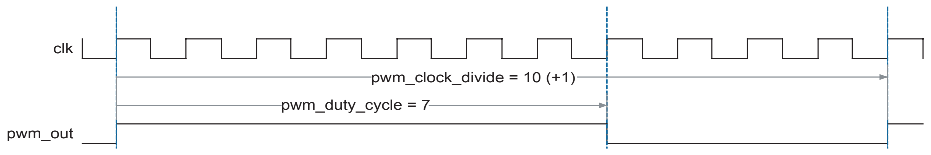
Table 10–2. PWM Design Files Directory

File Name	Description
/pwm_hw	Contains HDL files describing the component hardware.
pwm_task_logic.v	Contains the core of the PWM functionality.
pwm_register_file.v	Contains logic for reading and writing PWM registers.
pwm_avalon_interface.v	Instantiates task logic and register file, and provides an Avalon slave interface. This file contains the top-level module.
/pwm_sw	Contains C files describing the software interface to the component.
/inc	Contains header files defining low-level hardware interface.
avalon_slave_pwm_regs.h	Defines macros to access registers in the PWM component.
/HAL	Contains HAL driver files for the Nios II processor.
/inc	Contains HAL driver include files.
altera_avalon_pwm_routines.h	Declares function prototypes for accessing the PWM.
/src	Contains HAL driver source code files.
altera_avalon_pwm_routines.c	Defines functions for accessing the PWM.
/test_software	Contains an example program to test the component hardware & software.
hello_altera_avalon_pwm.c	`main()` initializes the PWM hardware, and uses the PWM to blink an LED.

Functional Specification

A PWM component outputs a square wave with modulated duty cycle. A basic pulse-width waveform is shown in Figure 10–2.

Figure 10–2. Basic Pulse-Width Modulation Waveform



The PWM component is specified and created as follows:

- The task logic operates synchronously to a single clock.
- The task logic uses 32-bit counters to provide a suitable range of PWM periods and duty cycles.
- A host processor is responsible for setting the PWM period value and duty-cycle value. This requirement implies the need for a read/write interface to control logic.
- Register elements are defined to hold the PWM period value and duty-cycle value.
- The host processor can halt the PWM output by using an enable control bit.

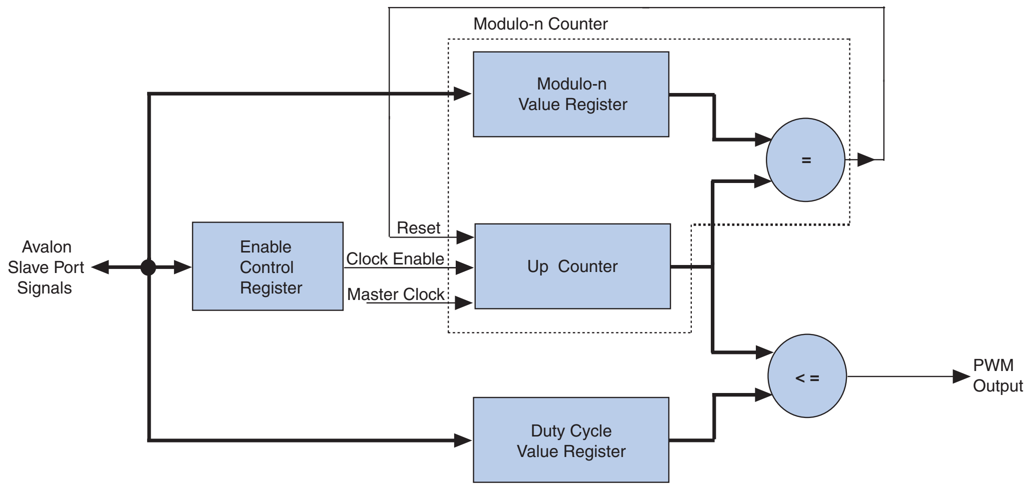
PWM Task Logic

The PWM task logic has the following characteristics:

- The PWM task logic consists of an input clock (`clk`), an output signal (`pwm_out`), an enable bit, a 32-bit modulo-n counter, and a 32-bit comparator circuit.
- `clk` drives the 32-bit modulo-n counter to establish the period of the `pwm_out` signal.
- The comparator compares the current value of the modulo-n counter to the duty-cycle value and determines the output of `pwm_out`.
- When the current count value is less than or equal to the duty-cycle value, `pwm_out` drives logic value 0; otherwise, it drives logic value 1.

The task-logic structure is shown in Figure 10–3.

Figure 10–3. PWM Task Logic Structure



Register File

The register file provides access to the enable bit, the modulo-n value and the duty cycle value, shown in Figure 10–3. The design maps each register to a unique offset in the Avalon slave port address space.

Each register has read and write access, which means that software can read back values previously written into the registers. This is an arbitrary design choice that provides software convenience at the expense of hardware resources. You could equally design the registers to be write-only, which would conserve on-chip logic resources, but make it impossible for software to read back the register values.

The register file and offset mapping is shown in Table 10–3. To support three registers, two bits of address encoding are necessary. This gives rise to the fourth register which is reserved.

Table 10–3. Register File & Address Mapping

Register Name	Offset	Access	Description
`clock_divide`	00	Read / Write	The number of clock cycles counted during one cycle of the PWM output.
`duty_cycle`	01	Read / Write	The number of clock cycles in which the PWM output will be low.
`enable`	10	Read / Write	Enables/disables the PWM output. Setting bit `0` to `1` enables the PWM.
Reserved	11	-	

To read or write the registers requires only one clock cycle, which affects the wait-states for the Avalon interface.

Avalon Interface

The Avalon interface for the PWM component requires a single slave port using a small set of Avalon signals to handle simple read and write transfers to the registers. The component's Avalon slave port has the following characteristics:

- It is synchronous to the Avalon slave port clock.
- It is readable and writeable.
- It has zero wait states for reading and writing, because the registers are able to respond to transfers within one clock cycle.
- It has no setup or hold restrictions for reading and writing.
- Read latency is not required, because all transfers can complete in one clock cycle. Read latency would not improve performance.
- It uses native address alignment, because the slave port is connected to registers rather than a memory device.

Table 10–4 lists the Avalon signals types required to implement these transfer properties. The table also lists the names of each signal as defined in the HDL design file.

Table 10–4. PWM Signal Names & Avalon Signal Types

Signal Name in HDL	Avalon Signal Type	Bit-Width	Direction	Notes
`clk`	`clk`	1	input	Clock that synchronizes data transfers and task logic
`resetn`	`reset_n`	1	input	Reset signal; active low.
`avalon_chip_select`	`chipselect`	1	input	Chip-select signal
`address`	`address`	2	input	2-bit address; only three encodings are used.
`write`	`write`	1	input	Write enable signal
`write_data`	`writedata`	32	input	32-bit write-data value
`read`	`read`	1	input	Read enable signal
`read_data`	`readdata`	32	output	32-bit read-data value

For details on the behavior of Avalon signals and Avalon transfers, see the *Avalon Interface Specification*.

Software API

The PWM example design provides both a header file that defines the register map, and driver software for the Nios II processor. See Table 10–2 on page 10–10 for a description of the individual files. The driver functions are listed in Table 10–5.

Table 10–5. PWM Driver Functions (1)

Function	Prototype Description
`altera_avalon_pwm_init();`	Initializes the PWM hardware
`altera_avalon_pwm_enable();`	Activates the PWM output
`altera_avalon_pwm_disable();`	Deactivates the PWM output
`altera_avalon_pwm_change_duty_cycle();`	Changes the PWM duty cycle

Note for Table 10–5:
(1) Each function takes a parameter that specifies the base address of a specific instance of the PWM component.

Package the Design Files into an SOPC Builder Component

In this section, you will use the SOPC Builder component editor to package the design files into an SOPC Builder component. You will perform the following operations:

1. Open the Quartus II project and start the component editor.
2. Configure the settings on each tab of the component editor.
3. Save the Component.

Open the Quartus II Project & Start the Component Editor

To open SOPC Builder from the Quartus II software, you must have a Quartus II project open. 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**. The SOPC Builder GUI appears, displaying a ready-made example design containing a Nios II processor and several components in the table of active components.
4. On the File menu, click **New Component**. The component editor GUI appears, displaying the **Introduction** tab.

HDL Files Tab

In this section you will associate the HDL files with the component using the **HDL Files** tab. Perform the following steps:

1. Click the **HDL Files** tab.

☞ Each tab in the component editor GUI provides on-screen information that describes how to use each tab. Click the triangle at the top-left of each tab to view these instructions.

2. Click **Add HDL File**.
3. Browse to the <*PWM design files*>**/pwm_hw** directory. There are three Verilog HDL (**.v**) files in this directory.

4. Select all three HDL files in this directory and click **Open**. Use the control key to select multiple files.

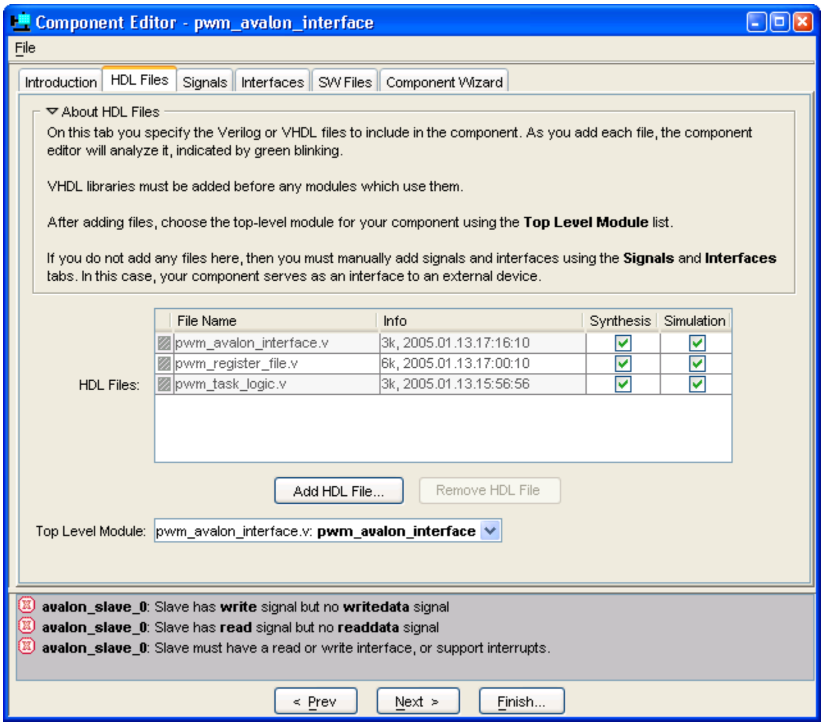
   You will return to the **HDL Files** tab. The component editor immediately analyzes each file to read I/O signal and parameter information from the file.

5. Ensure that both the **Simulation** and **Synthesis** boxes are turned on for all files. This indicates that each file is appropriate for both simulation and synthesis design flows.

6. Select **pwm_avalon_interfave.v: pwm_avalon_interface** in the **Top Level Module** list to specify the top-level module.

At this point, the component editor GUI displays error messages. Ignore these messages for now, because you will fix them in later steps. Figure 10–4 shows the state of the **HDL Files** tab.

Figure 10–4. HDL Files Tab



On this tab you specify the Verilog or VHDL files to include in the component. As you add each file, the component editor will analyze it, indicated by green blinking.

VHDL libraries must be added before any modules which use them.

After adding files, choose the top-level module for your component using the **Top Level Module** list.

If you do not add any files here, then you must manually add signals and interfaces using the **Signals** and **Interfaces** tabs. In this case, your component serves as an interface to an external device.

HDL Files:

File Name	Info	Synthesis	Simulation
pwm_avalon_interface.v	3k, 2005.01.13.17:16:10	✓	✓
pwm_register_file.v	6k, 2005.01.13.17:00:10	✓	✓
pwm_task_logic.v	3k, 2005.01.13.15:56:56	✓	✓

Top Level Module: pwm_avalon_interface.v: **pwm_avalon_interface**

avalon_slave_0: Slave has **write** signal but no **writedata** signal
avalon_slave_0: Slave has **read** signal but no **readdata** signal
avalon_slave_0: Slave must have a read or write interface, or support interrupts.

Signals Tab

For every I/O signal present on the top-level HDL module, you must map the signal name to a valid Avalon signal type using the **Signals** tab. The component editor automatically fills in signal details that it finds in the top-level HDL source file. If a signal is named the same as a recognized Avalon signal type (such as `write` or `address`), then the component editor automatically assigns the signal's type. If the component editor cannot determine the signal type, it assigns it to type **export.**

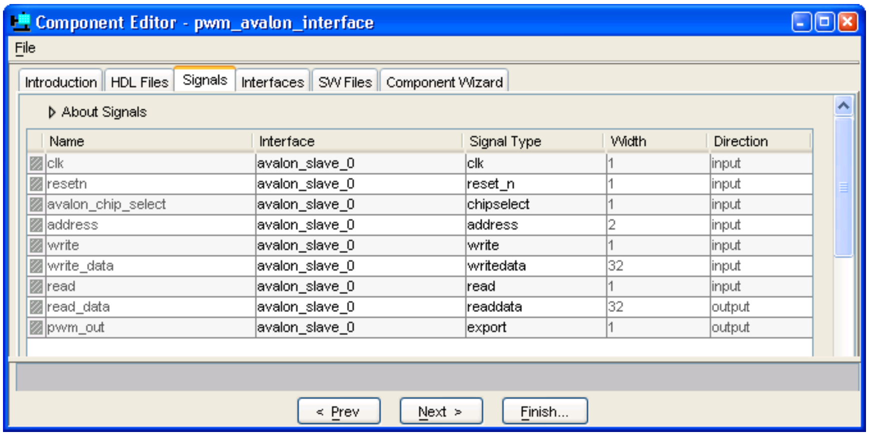
Perform the following steps to define the component I/O signals:

1. Click the **Signals** tab. All of the I/O signals in the top level HDL module **pwm_avalon_interface** appear automatically.

2. Assign the **Signal Type** settings for all signals, as show in Figure 10–5. To change a value, click the **Signal Type** cell to display a drop-down list, and select a new signal type from the list.

After you correctly assign each signal name to a signal type, the error messages should disappear.

Figure 10–5. Assigning Signal Names to Signal Types



Name	Interface	Signal Type	Width	Direction
clk	avalon_slave_0	clk	1	input
resetn	avalon_slave_0	reset_n	1	input
avalon_chip_select	avalon_slave_0	chipselect	1	input
address	avalon_slave_0	address	2	input
write	avalon_slave_0	write	1	input
write_data	avalon_slave_0	writedata	32	input
read	avalon_slave_0	read	1	input
read_data	avalon_slave_0	readdata	32	output
pwm_out	avalon_slave_0	export	1	output

You assign type **export** to the signal `pwm_out`, because it is not an Avalon signal. It is intended to be an output of the SOPC Builder system.

Interfaces Tab

The **Interfaces** tab lets you configure the properties of all Avalon interfaces on the component. In this case there is only one Avalon interface, as specified in the section "Avalon Interface" on page 10–13. Perform the following steps to configure the Avalon slave port:

1. Click the **Interfaces** tab. The component editor displays a default Avalon slave port that it created automatically, based on the top-level I/O signals in the component design.

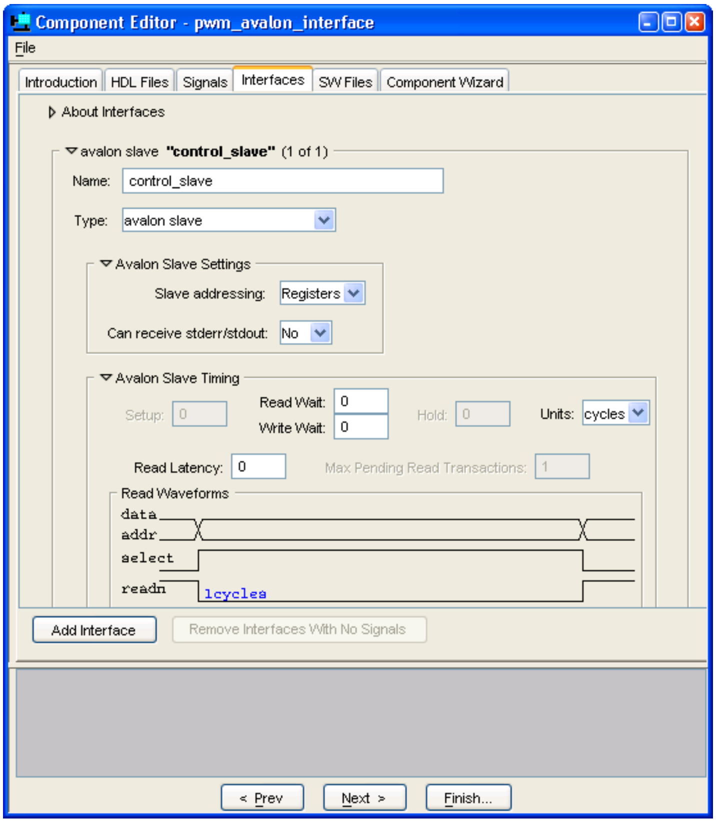
2. Type `control_slave` in the **Name** field to rename the slave port. This name appears in the SOPC Builder GUI when you instantiate the component in SOPC Builder.

3. Change the settings for the **control_slave** interface as listed in Table 10–6 below. Figure 10–6 on page 10–19 shows the **Interfaces** tab with the correct settings.

Table 10–6. Control Slave Interface Settings

Setting	Value	Description
Slave addressing	Registers	This setting is appropriate for slave ports used to access address-mapped registers
Read Wait	0	This setting means that the slave port responds to read requests in a single clock cycle (that is, it does not need read waitstates.)
Write Wait	0	This setting means that the slave port captures write requests in a single clock cycle (that is, it does not need write waitstates.)

Figure 10–6. Configuring the Interface Properties


Component Editor - pwm_avalon_interface

File

Introduction | HDL Files | Signals | Interfaces | SW Files | Component Wizard

About Interfaces

avalon slave **"control_slave"** (1 of 1)

Name: control_slave

Type: avalon slave

Avalon Slave Settings

Slave addressing: Registers

Can receive stderr/stdout: No

Avalon Slave Timing

Setup: 0 | Read Wait: 0 | Write Wait: 0 | Hold: 0 | Units: cycles

Read Latency: 0 | Max Pending Read Transactions: 1

Read Waveforms

Software Files (SW Files) Tab

The **SW Files** tab lets you associate software files with the component, and specify their usage. This component example design provides both a header file that defines the registers and driver software for the Nios II processor. For a description of each file, see Table 10–2 on page 10–10.

Perform the following steps to import the software files into the component:

1. Click the **SW Files** tab.

2. Click **Add SW File**. The Open dialog appears.

3. Browse to the directory <*PWM design files*>**/pwm_sw/inc**.

4. Select the file **altera_avalon_pwm_regs.h** and click **Open**.

5. Click the **Type** cell for **altera_avalon_pwm_regs.h** to change the file type. A drop-down list appears.

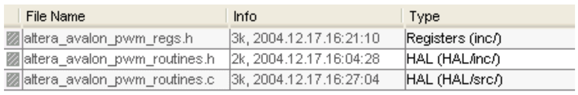
6. Select **type Registers (inc/)**.

7. Repeat steps 2 to 6 to add the file <*PWM design files*>**/pwm_sw/HAL/inc/altera_avalon_pwm_routines.h** and set its type to **HAL (HAL/inc/)**.

8. Repeat steps 2 to 6 to add the file <*PWM design files*>**/pwm_sw/HAL/src/altera_avalon_pwm_routines.c** and set its type to **HAL (HAL/src/)**.

Figure 10–7 shows the SW Files tab with the correct settings.

Figure 10–7. Software Files (SW Files) Tab



	File Name	Info	Type
▨	altera_avalon_pwm_regs.h	3k, 2004.12.17.16:21:10	Registers (inc/)
▨	altera_avalon_pwm_routines.h	2k, 2004.12.17.16:04:28	HAL (HAL/inc/)
▨	altera_avalon_pwm_routines.c	3k, 2004.12.17.16:27:04	HAL (HAL/src/)

Component Wizard Tab

This tab lets you control how SOPC Builder presents the component to a user. Perform the following steps to configure the user presentation of the component:

1. Click the **Component Wizard** tab.

2. For this example, do not change the default settings for **Component Name**, **Component Version**, and **Component Group**.

   These settings affect how SOPC Builder identifies the component and displays it in the list of available components. The component editor creates a default name for the component, based on the name of the top-level design module.

3. Under **Parameters**, in the **Tooltip** cell for the parameter **clock_divide_reg_init**, type the following:

   ```
   Initial PWM Period After Reset
   ```

4. In the **Tooltip** cell for **clock_cycle_reg_init**, type:

    ```
    Initial Duty Cycle After Reset
    ```

5. Click **Preview the Wizard** to preview how the component wizard will appear when instantiated from within SOPC Builder.

6. Close the preview window when you are done.

Save the Component

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

1. Click **Finish**. A dialog appears describing the files that will be created for the component.

2. Click **Yes** to save the files. The component editor saves the files to a subdirectory under **<*Quartus II project*>**. The component editor closes, and you return to the main SOPC Builder GUI.

3. Locate the new component **pwm_avalon_interface** in the list of available components under the **User Logic** group.

You are ready to instantiate the component into an SOPC Builder system.

Instantiate the Component in Hardware

At this point, the new component is ready to instantiate in an SOPC Builder system. The usage of a component is design dependent, based on the needs of the system. The remaining steps for this design example show one possible way to instantiate and test the component. However, there is an unlimited number of ways this component can be used in a system.

In this section you will add the new PWM component to a system, recompile the hardware design, and configure the FPGA. This section includes the following steps:

1. Add a PWM component to the SOPC Builder system and regenerate the system.

2. Modify the Quartus II design to connect the PWM output to an FPGA pin.

3. Compile the Quartus II design and configure the FPGA with the new hardware image.

Add a PWM Component to the SOPC Builder System

Perform the following steps to setup SOPC Builder's component search path:

1. In the SOPC Builder GUI, on the File menu, click **SOPC Builder Setup**.

2. Under **Component/Kit Library Search Path**, enter the path to the ***<Quartus II project>*** directory. If there are pre-existing paths, use "+" to separate the path names.

3. Click **OK**.

☞ The steps above make the component's software files visible to the Nios II IDE in later steps. These steps are necessary for the Quartus II software v4.2 and the Nios II IDE v1.1. Future releases will eliminate the need for these steps.

Perform the following steps to add a PWM component to the SOPC Builder system:

1. On the SOPC Builder **System Contents** tab, select the new component **pwm_avalon_interface** under the **User Logic** group in the list of available components, and click **Add**. The configuration wizard for the PWM component appears.

   If you want to, you can modify the parameters in the configuration GUI. The parameters affect the reset state of the PWM control registers, but have no affect on the outcome of the steps in this chapter.

2. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and the component **pwm_avalon_interface_0** appears in the table of active components.

3. Right-click **pwm_avalon_interface_0** and choose **Rename**.

4. Type `z_pwm_0` for the component name and press **Enter**. (This name is unusual, but it minimizes effort later when you update the Quartus II design in section "Modify the Quartus II Design to Use the PWM Output" on page 10–23.

⚠ You must name the component exactly as directed, or else later steps in this chapter will fail.

5. Click **Generate** to start generating the system.

6. After system generation completes successfully, exit SOPC Builder and return to the Quartus II software.

Modify the Quartus II Design to Use the PWM Output

At this point, you have created an SOPC Builder system that uses the PWM component. Now you must update the Quartus II project to use the PWM output.

The file **standard.bdf** is the top-level Block Design File (BDF) for the Quartus II project. The BDF contains a symbol for the SOPC Builder system module, named **std_<*FPGA*>**, where **<*FPGA*>** refers to the FPGA on the target development board.

In the previous steps you added a PWM component which produces an additional output from the system module. Now you need to update the symbol for the system module, and connect the PWM output to an FPGA pin.

☞ To complete this section, you must be familiar with the Quartus II Block Editor.

1. In the Quartus II software, open the file **standard.bdf**.

2. Right-click the symbol **std_<*FPGA*>** in the BDF and choose **Update Symbol or Block**. The Update Symbol or Block dialog appears.

3. Select **Selected Symbol(s) or Block(s)**.

4. Click **OK** to close the dialog. The symbol **std_<*FPGA*>** in the BDF is updated, and it now has an additional output port named **pwm_out_from_the_z_pwm_0**.

   ☞ SOPC Builder creates unique names for all I/O ports on the system module, by combining the signal name in the component design file with the instance name of the component in the system module.

5. Delete the symbol for pins `LEDG[7..0]` which are connected to port `out_port_from_the_led_pio[7..0]` on the system module.

   These pins connect to LEDs on the development board. This example design uses one of the LEDs to display the output of the PWM.

6. Create a new output pin named `LEDG[0]`.

7. Connect the new pin `LEDG[0]` to `pwm_out_from_the_z_pwm_0` on **std_<*FPGA*>**.

The hardware design is now ready to compile.

Compile the Hardware Design & Download to the Target Board

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

1. On the File menu, click **Save** to save changes to the BDF.

2. 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 will finish successfully after several minutes, and generate a new FPGA configuration file for the project.

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

Perform the following steps to download the hardware design to the board:

1. Connect your host computer to the development board using an Altera download cable, such as the USB Blaster, and apply power to the board.

2. On the Tools menu, click **Programmer** to open the Quartus II Programmer.

3. 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 PWM example design is based on the Nios II processor. You must execute software on the Nios II processor to exercise the PWM hardware. The example design files provide a C test program that pulses an LED by gradually modulating the PWM duty cycle. This test program accesses the hardware both by using the register map declarations directly, and by calling the driver functions.

In this section you will 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.

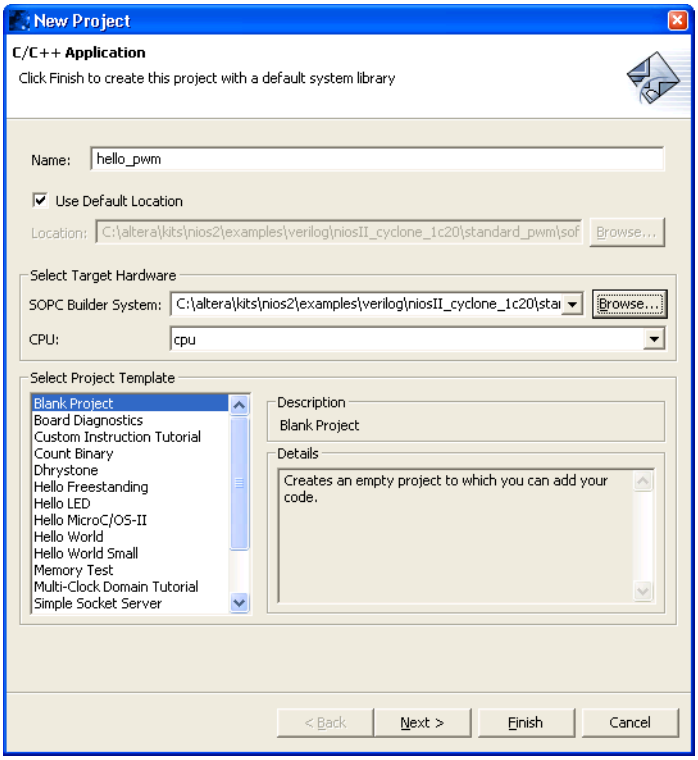
To complete this section, you must have performed all prior steps, and successfully configured the target board with the hardware design.

Start the Nios II IDE & 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 File menu, click **New > C/C++ Application** to start a new project. The first page of the **New Project** wizard appears.
3. Under **Select Project Template**, select **Blank Project**.
4. In the **Name** field, type `hello_pwm`.
5. Ensure that **Use Default Location** is turned on.
6. Click **Browse** under **Select Target Hardware**. The **Select Target Hardware** dialog box appears.
7. Browse to the **<*Quartus II project*>** directory.
8. Select the file **std_<*FPGA*>.ptf**.
9. Click **Open** to return to the New Project wizard. The **SOPC Builder System** and the **CPU** fields are now specified, as shown in Figure 10–8 on page 10–26.
10. Click **Finish**.

Figure 10–8. New Project Wizard



Name: hello_pwm

[✓] Use Default Location

Location: C:\altera\kits\nios2\examples\verilog\niosII_cyclone_1c20\standard_pwm\sof

SOPC Builder System: C:\altera\kits\nios2\examples\verilog\niosII_cyclone_1c20\sta

CPU: cpu

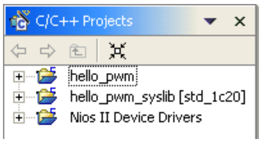
Blank Project
Board Diagnostics
Custom Instruction Tutorial
Count Binary
Dhrystone
Hello Freestanding
Hello LED
Hello MicroC/OS-II
Hello World
Hello World Small
Memory Test
Multi-Clock Domain Tutorial
Simple Socket Server

Description: Blank Project

Details: Creates an empty project to which you can add your code.

After the IDE successfully creates the new project, the C/C++ Projects view will contain two new projects, **hello_pwm** and **hello_pwm_syslib**, in addition to **Nios II Device Drivers**, as shown in Figure 10–9.

Figure 10–9. New Projects in the C/C++ Projects View



Compile the Software Project & Run on the Target Board

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

First, perform the following steps to associate the C source file with the new C/C++ project.

1. In your computer's file system, copy the file *<PWM design files>***/pwm_sw/test_software/hello_altera_avalon_pwm.c** to the directory ***<Quartus II project>*****/software/hello_pwm/**.

2. In the Nios II IDE C/C++ Projects view, right-click **hello_pwm** and choose **Refresh**. This forces 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 **hello_pwm** and choose **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 **hello_pwm** in the C/C++ Projects view.

3. On the Run menu, click **Run**. The Run dialog appears.

4. Under **Configurations** select **Nios II Hardware**, and click **New**. A new run/debug configuration named **hello_pwm Nios II HW configuration** appears.

5. If the **Run** button (in the bottom right of the Run dialog) is deactivated, perform the following steps:

   a. Click the **Target Connection** tab.

   b. Click **Refresh** next to the **JTAG cable** list.

   c. From the **JTAG cable** list, select the download cable you want to use.

   d. Click **Refresh** next to the **JTAG device** list.

6. Click **Run**.

7. View the results:

   a. The **Console** view in the IDE displays messages similar to the following:

```
Hello from the PWM test program.
The starting values in the PWM registers are:
Period = 0
Duty cycle = 0
Notice the pulsing LED on the development board.
```

   b. LED0 on the development board repeatedly pulses on and off.

Congratulations! You have finished all steps for the PWM design example.

Sharing Components

When you create a component using the component editor, SOPC Builder automatically saves the component in the current Quartus II project directory. 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:

1. 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.

   The directory path name cannot contain spaces. If the path contains spaces, SOPC Builder might not be able to access the files.

2. In SOPC Builder, on the File menu, click **SOPC Builder Setup**. The **SOPC Builder Setup** dialog appears, which lets you specify where SOPC Builder searches for component files.

3. Under **Component/Kit Library Search Path**, add the path to the enclosing directory of the component directory. For example, for a component directory **c:\my_component_library\pwm_avalon_interface\**, add the path **c:\my_component_library**. If there are pre-existing paths, use "+" to separate the path names.

4. Click **OK**.

11. Building Systems with Multiple Clock Domains

QII54008-6.0.0

Introduction

This chapter guides you through the process of using SOPC Builder to create a system with multiple clock domains. You will start with a ready-made design that uses a single clock domain, and modify the design to use two clocks.

Example Design Overview

The design in this chapter mimics the common scenario of a system with separate control and data paths. Typically, the control path is slow, because the controller itself is relatively slow, and it is used only in short bursts to set up data transfers. On the other hand, the data path is fast so that, after the controller initiates a transfer, data moves as quickly as possible from source to destination.

Figure 11–1 on page 11–2 shows a simplified block diagram of the system structure. In this design, a Nios® II processor acts as the controller operating at 50 MHz. A DMA controller operating at 100 MHz manages the data path, and reads and writes data buffers that also operate at 100 MHz. The figure focuses on the multi-clock nature of the system, and shows the connections between master and slave ports.

Figure 11–1. Simplified Block Diagram of the Example Design

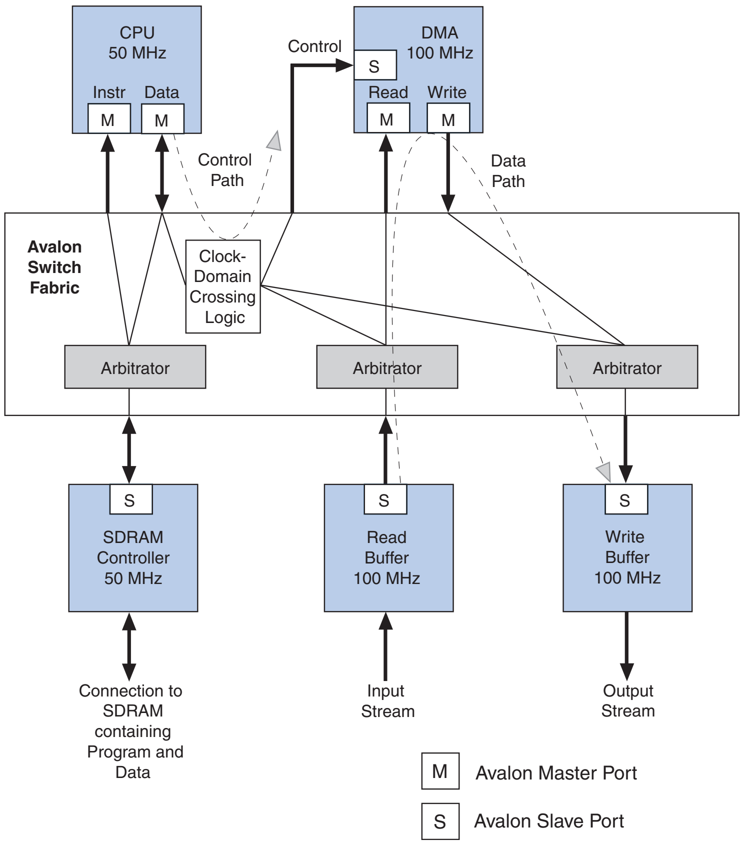


Figure 11–1 does not show example design features that are not directly related to the issue of multiple clocks, such as a JTAG UART communication peripheral and timer peripheral used by the Nios II processor.

Hardware & Software Requirements

To use the design example(s) in this chapter, you must have the following:

- Design files for the example design – A hyperlink to the design files is located with this chapter on the SOPC Builder literature page. Visit **www.altera.com/sopcbuilder**.

- Quartus® II Software version 4.2 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® II Edition
  - Stratix Edition
  - Stratix Professional Edition
  - Cyclone™ Edition

  If you do not have a development board, you can follow the hardware development steps, but you will not be able to 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**.

Before you begin, you must install the Quartus II software and Nios II EDS.

Creating the Multi-Clock Hardware System

In this section, you will start with an example hardware design provided with the Nios II EDS and create a multi-clock system. You will perform the following steps:

1. Copy the hardware design files to a new directory

2. Modify the design in SOPC Builder to create a multi-clock hardware system

3. Update the Quartus II design to use the new clock domain

4. Compile the hardware design in the Quartus II software, and download the hardware design to a target board

Copy the Hardware Design Files to a New Directory

The hardware design used in this chapter is based on the **standard** hardware example design included with the Nios II EDS. Copy the design files by performing the following steps:

1. In 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 version of the design. You can use either one. Table 11–1 shows the names of the directories for the available Nios development boards.

Table 11–1. Design File Directories

Nios Development Board	Tutorial Directory
Stratix II Edition	niosII_stratixII_2s60_es
Stratix Edition	niosII_stratix_1s10 or niosII_stratix_1s10_es
Stratix Professional Edition	niosII_stratix_1s40
Cyclone Edition	niosII_cyclone_1c20

   For demonstration purposes, the figures in this chapter show the case of the Verilog design on the Nios Development Board, Cyclone Edition.

2. Copy the **standard** directory to a new location. This document will refer to the newly-created directory as *<hardware files directory>*.

Modify the Design in SOPC Builder

This section walks you through the process of implementing a multi-clock system in SOPC Builder. In this section you will do the following:

1. Open the system in SOPC Builder.
2. Add a DMA controller and two 4 Kbyte on-chip memory components.
3. Connect DMA master ports to memory slave ports.
4. Make clock domain assignments.
5. Regenerate the system.

Open the System in SOPC Builder

To open the system in SOPC Builder, perform the following steps:

1. Start the Quartus II software.

2. Choose **Open Project** (File menu).

3. Browse to <*hardware files directory*>.

4. Select **standard.qpf** and click **Open**.

5. Choose **SOPC Builder** (Tools menu) to start SOPC Builder.

The SOPC Builder window appears, displaying the contents of the system, which you will use as a starting point for your design.

Add DMA Controller & Memory Components

Perform the following steps to add the DMA controller and memory components to the system:

1. In the SOPC Builder list of available components, select **DMA** in the **Other** group, and click **Add**. The DMA configuration wizard displays.

2. In the DMA configuration wizard, click **Finish** to accept the default settings. You return to the SOPC Builder **System Contents** tab which displays the new DMA component, named **dma_0**.

   Errors are displayed in the SOPC Builder messages window. You can ignore these messages for now, because you will fix the errors in later steps.

3. In the list of available components, select **On-Chip Memory (RAM or ROM)** in the **Memory** group, and click **Add**. The On-Chip Memory configuration wizard appears.

4. In the On-Chip Memory configuration wizard, click **Finish** to accept the default settings. You return to the SOPC Builder **System Contents** tab, which displays the new memory component.

5. Right-click the new memory component and choose **Rename**.

6. Type `read_buffer`↵ to rename the component.

7. Repeat steps 3 and 4 to add another On-Chip Memory component to the system.

8. Right-click the new memory component and choose **Rename**.

9. Type `write_buffer`↵ to rename the component.

⚠ You must name the DMA and memory components exactly as specified above (**dma_0, read_buffer**, and **write_buffer**). If you name the components differently, later steps will fail.

Connect DMA Master Ports to Memory Slave Ports

Now that you have added the DMA controller and the memory components to the system, you must connect their master and slave ports appropriately. Perform the following steps:

1. Hover the mouse pointer over the connections panel in the SOPC Builder **System Contents** tab to display the potential master port connections for the DMA. See Figure 11–2.

2. Connect the DMA read master port (**dma_0/read_master**) to the **read_buffer** memory.

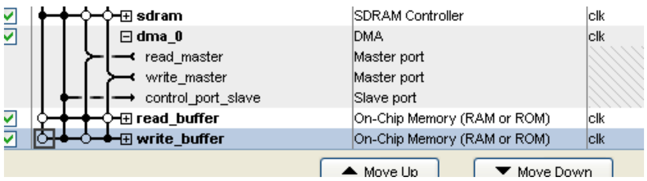
3. Connect the DMA write master port (**dma_0/write_master**) to the **write_buffer** memory.

4. Disconnect the Nios II processor instruction master (**cpu/instruction_master**) from both the on-chip memories' slave ports. For this example design, the processor does not use these memories to fetch instructions.

5. Verify that the Nios II processor data master (**cpu/data_master**) is connected to the DMA slave port (**dma/control_port_slave**).

Figure 11–2 shows the state of the connections panel after all components have been correctly connected.

Figure 11–2. Correct Connections Between Master & Slave Ports



✓ ⊞ **sdram**	SDRAM Controller	clk
✓ ⊟ **dma_0**	DMA	clk
read_master	Master port	
write_master	Master port	
control_port_slave	Slave port	
✓ ⊞ **read_buffer**	On-Chip Memory (RAM or ROM)	clk
✓ ⊞ **write_buffer**	On-Chip Memory (RAM or ROM)	clk

After the DMA controller is connected properly to the on-chip memories, the error messages disappear from the SOPC Builder messages window.

Make Clock Domain Assignments

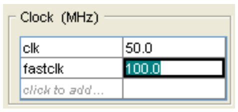
In this section you will specify a 100 MHz clock input, and assign the DMA and memory components to the new clock domain. Then you will generate the system. Perform the following steps:

1. On the **System Contents** tab under **Clock MHz**, click the cell labeled **click to add** to enter a new clock entry.

2. Type **fastclk** for the name of the new clock, as shown in Figure 11–3.

3. Type **100** for the speed of `fastclk`, as shown in Figure 11–3.

Figure 11–3. Clock Settings



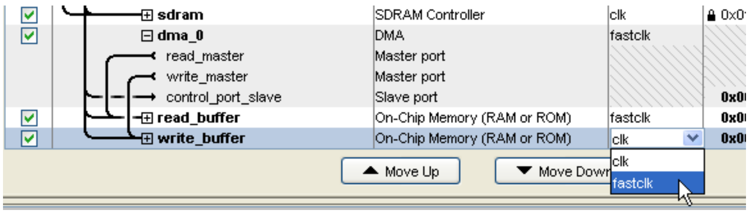
Clock (MHz)

clk	50.0
fastclk	100.0
click to add...	

4. In the **Clock** list for the DMA component, select **fastclk** to assign the 100 MHz clock to the DMA component, as shown in Figure 11–4.

5. Repeat step 4 for the **read_buffer** and **write_buffer** components, as shown in Figure 11–4.

Figure 11–4. Assigning a Different Clock to the DMA & Memory Components



✓	sdram	SDRAM Controller	clk	0x0
✓	dma_0	DMA	fastclk	
	read_master	Master port		
	write_master	Master port		
	control_port_slave	Slave port		0x0
✓	read_buffer	On-Chip Memory (RAM or ROM)	fastclk	0x0
✓	write_buffer	On-Chip Memory (RAM or ROM)	clk	0x0

6. Click **Generate** to start generating the system.

7. After system generation completes successfully, exit SOPC Builder and return to the Quartus II software.

Update the Quartus II Design

At this point, you have created an SOPC Builder system that uses multiple clock domains. In this section you will do the following:

1. Update the system module symbol.
2. Update PLL settings to generate a 100 MHz clock.
3. Connect the 100 MHz clock to the system module.
4. Compile the Design and download it to the board.

To complete this section, you must be familiar with the Quartus II Block Editor.

Update the System Module Symbol

The file **standard.bdf** is the top-level Block Diagram File (BDF) for the Quartus II project. The BDF contains a symbol for the SOPC Builder system module, named **std_*<FPGA>***, where ***<FPGA>*** refers to the FPGA on the target development board.

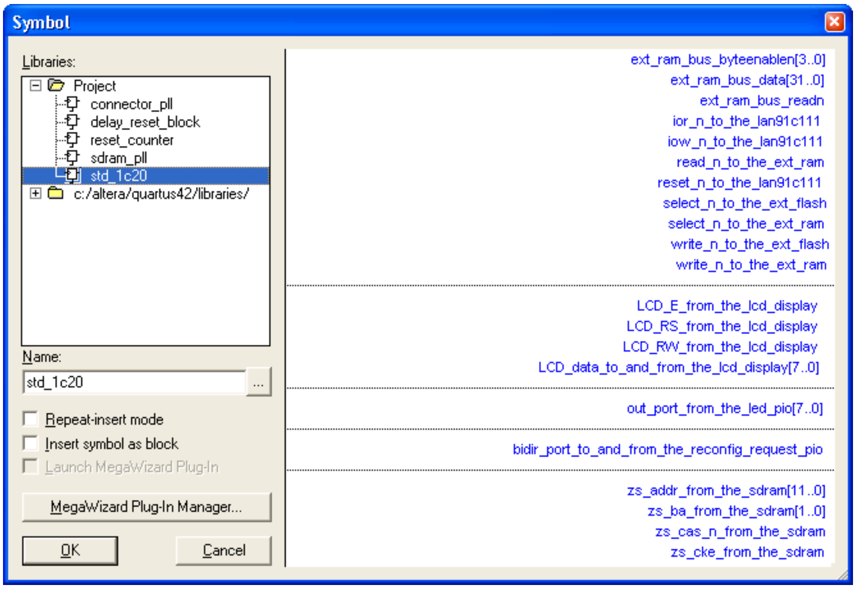
The SOPC Builder system module requires an additional clock input for the 100 MHz clock domain, and therefore you need to update the symbol for the system module. To remove the old symbol and insert an updated symbol, perform the steps below.

In the following steps you will disconnect and then reconnect all connections to the system module, including connections to FPGA pins. You must make these connections exactly, or else the design will not work in hardware, potentially damaging the development board. If possible, print the BDF as a reference to help you reconnect pins to the system module later.

1. In the BDF, select the symbol **std_*<FPGA>*** and delete it.
2. Click-and-drag to select all of the pins that previously connected to the right-hand-side of the symbol.
3. Press the right-arrow key ten times to move the pins to the right, creating space for the new symbol.
4. Double click in the space where the symbol used to be to insert a new symbol. The **Symbol** window appears.
5. Expand the **Project** folder under **Libraries**.

6. Select the **std_<*FPGA*>** symbol, and click OK. See Figure 11–5.

Figure 11–5. Selecting & Inserting the New Symbol


Symbol
Libraries:
Project
connector_pll
delay_reset_block
reset_counter
sdram_pll
std_1c20
c:/altera/quartus42/libraries/
Name:
std_1c20
Repeat-insert mode
Insert symbol as block
Launch MegaWizard Plug-In
MegaWizard Plug-In Manager...
OK
Cancel
ext_ram_bus_byteenablen[3..0]
ext_ram_bus_data[31..0]
ext_ram_bus_readn
ior_n_to_the_lan91c111
iow_n_to_the_lan91c111
read_n_to_the_ext_ram
reset_n_to_the_lan91c111
select_n_to_the_ext_flash
select_n_to_the_ext_ram
write_n_to_the_ext_flash
write_n_to_the_ext_ram
LCD_E_from_the_lcd_display
LCD_RS_from_the_lcd_display
LCD_RW_from_the_lcd_display
LCD_data_to_and_from_the_lcd_display[7..0]
out_port_from_the_led_pio[7..0]
bidir_port_to_and_from_the_reconfig_request_pio
zs_addr_from_the_sdram[11..0]
zs_ba_from_the_sdram[1..0]
zs_cas_n_from_the_sdram
zs_cke_from_the_sdram

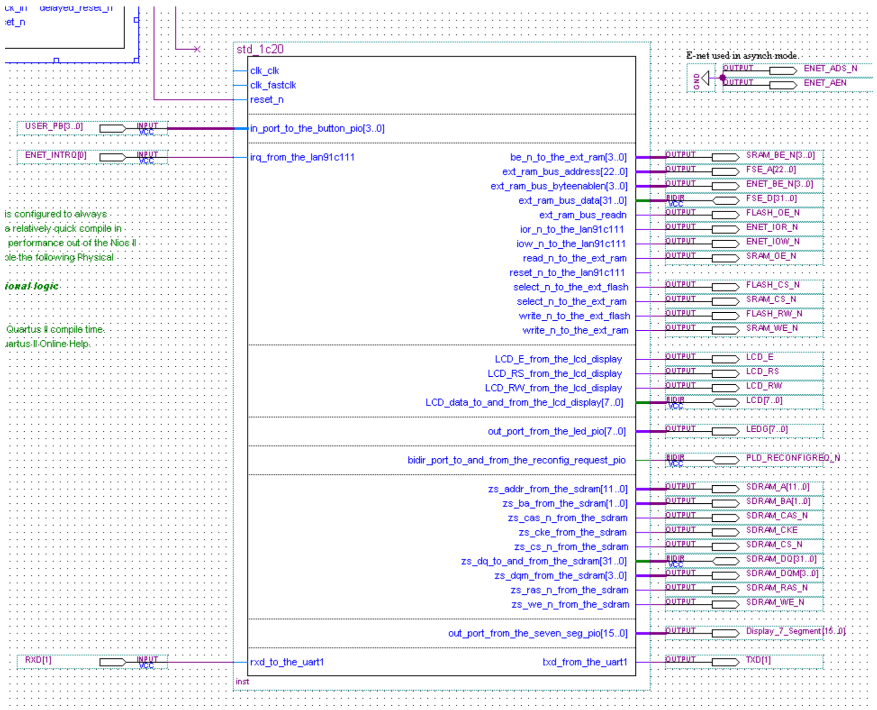
7. Move the mouse to position the symbol between the pin symbols, then click the left mouse button to place the symbol.

8. Look at the symbol, and notice the new clock input, **clk_fastclk**, and the old clock input, which is now named **clk_clk**.

9. Reconnect all previous connections to the **std_<*FPGA*>** symbol, except for the clock inputs **clk_clk** and **clk_fastclk**. You will connect the clock inputs in later steps.

Figure 11–6 shows the new symbol reconnected to all signals except for the clocks.

Figure 11–6. Updated Symbol Without Clock Connections



Update PLL Settings to Generate a 100 MHz Clock

Perform the following steps to modify the PLL instance in the BDF to generate a 100 MHz clock:

1. Locate the symbol **sdram_pll** in the BDF.

2. Right-click the symbol and select **MegaWizard® Plug-in Manager** to configure the PLL settings. The MegaWizard Plug-In Manager for the ALTCLKLOCK function displays.

   At this point, the Quartus II software might display a benign warning: `Delay shifts (time delay elements) are no longer supported in the Stratix PLLs`. If you see this message, click **OK**.

3. Click **Next** until you reach page 6 of 15 of the wizard flow, shown in Figure 11–7.

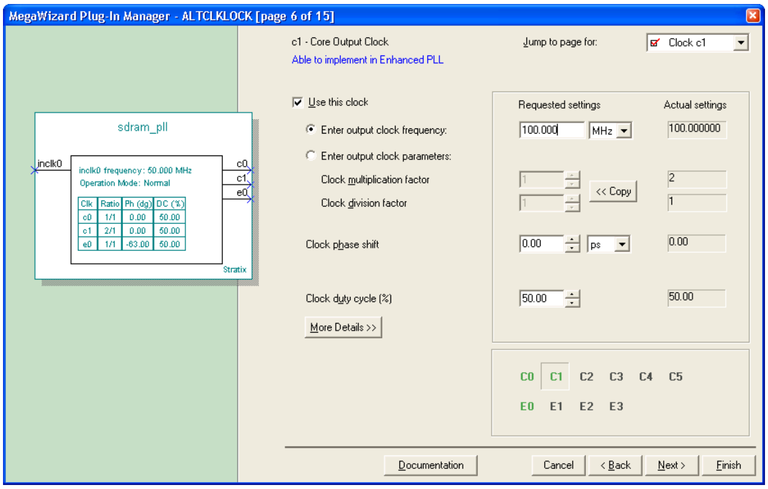
4. Click **Use this clock**.

5. Select **Enter output clock frequency**.

6. Under **Requested settings,** type `100` and select **MHz**.

Figure 11–7 shows the **ALTCLKLOCK** MegaWizard Plug-In Manager with the correct settings.

Figure 11–7. ALTCLKLOCK Settings to Generate a 100MHz Clock



inclk0 frequency: 50.000 MHz
Operation Mode: Normal

Clk	Ratio	Ph (dg)	DC (%)
c0	1/1	0.00	50.00
c1	2/1	0.00	50.00
e0	1/1	-63.00	50.00

c1 - Core Output Clock
Able to implement in Enhanced PLL

☑ Use this clock

	Requested settings	Actual settings
● Enter output clock frequency:	100.000 MHz	100.000000
○ Enter output clock parameters:		
Clock multiplication factor	1	2
Clock division factor	1	1
Clock phase shift	0.00 ps	0.00
Clock duty cycle (%)	50.00	50.00

7. Click **Finish** to jump to the final stage (page 15 of 15) of the wizard flow.

8. Click **Finish** again to return to the Quartus II software.

9. Right-click the **sdram_pll** symbol and choose **Update Symbol or Block**. The Update Symbol or Block dialog appears.

10. Click **OK** to update the selected instance of the symbol.

The PLL is now configured correctly to generate a 100 MHz output clock.

For further information on using the PLLs in Altera devices, see the handbook for the target device family.

Connect the 100 MHz Clock to the System Module

You now must connect the clock signals to the SOPC Builder system module. The easiest way to accomplish this is to symbolically link the PLL outputs to the system module inputs using conduit aliases. This section will guide you through the process of making one connection, and leave the remaining connections to you as an exercise.

To connect the 100 MHz clock signal from the PLL to the system module, perform the following steps:

1. Move the mouse pointer over node **c1** on **sdram_pll** until the pointer changes to a cross-hairs.
2. Click and drag right to add a conduit (i.e. a connection line) to node **c1**.
3. Click on the conduit line to select it.
4. Type `dmaclk` and press Enter.
5. Move the mouse pointer over node **clk_fastclk** on **std_<FPGA>** until the pointer changes to a cross-hairs.
6. Click and drag left to add a conduit to node **clk_fastclk**.
7. Click on the conduit line to select it.
8. Type `dmaclk` and press **Enter**.

The two nodes are now linked symbolically (by the name `dmaclk`) via a conduit alias.

Follow the instructions below to complete the remaining clock connections. Depending on which Nios development board you are targeting, the remaining PLL connections to the PLL(s) are slightly different. This section gives instructions to accommodate all Nios development boards.

Be sure to use the instructions that apply to your board, or else the design will fail in hardware.

For the Nios Development Board, Stratix II Edition, Stratix Professional Edition, and Stratix Edition, connect the PLL according to Table 11–2.

Table 11–2. PLL Connections

Node on sdram_pll	Connects to
c0	`clk_clk` on **std_*<FPGA>***
c1	`clk_fastclk` on **std_*<FPGA>***
e0	`PLD_CLKOUT` pin

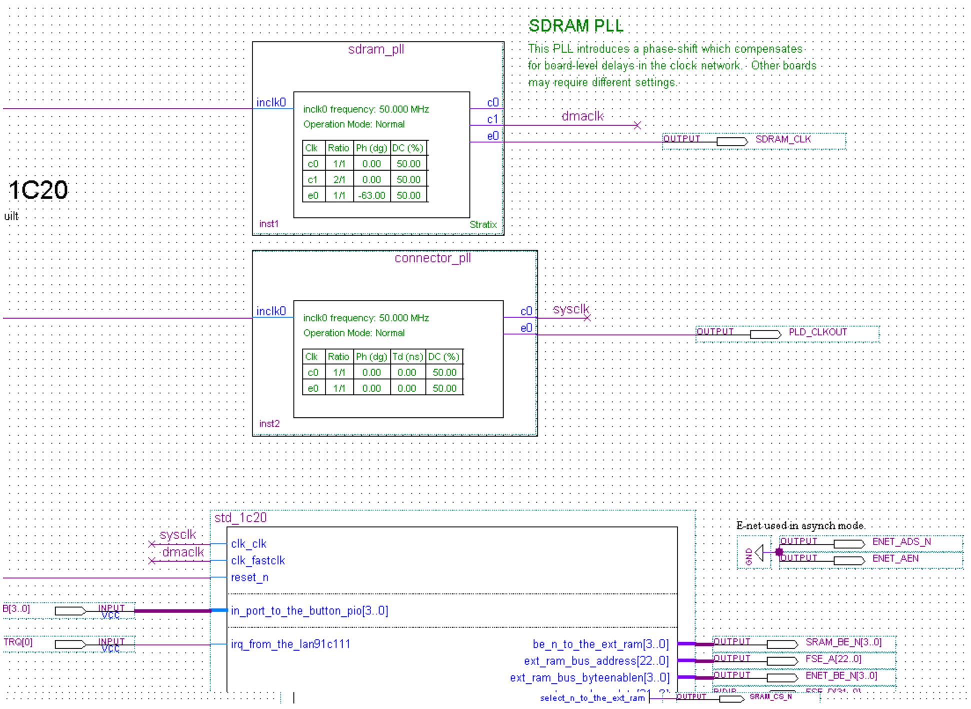
The BDF for the Nios Development Board, Cyclone Edition contains a second PLL symbol named **connector_pll** in addition to **sdram_pll**. Connect both PLLs according to Table 11–3.

Table 11–3. PLL Connections for the Nios Development Board, Cyclone Edition

Node	Connects to
c0 on **sdram_pll**	Nothing
c1 on **sdram_pll**	`clk_fastclk` on **std_*<FPGA>***
e0 on **sdram_pll**	`SDRAM_CLK` pin
c0 on **connector_pll**	`clk_clk` on **std_*<FPGA>***
e0 on **connector_pll**	`PLD_CLKOUT` pin

Figure 11–8 shows an example of the BDF for the Nios Development Board, Cyclone Edition with all connections completed using conduit aliases.

Figure 11–8. Symbolic Conduit Aliases Connecting PLL(s) to System Module



inclk0 frequency: 50.000 MHz
Operation Mode: Normal

Clk	Ratio	Ph (dg)	DC (%)
c0	1/1	0.00	50.00
c1	2/1	0.00	50.00
e0	1/1	-63.00	50.00

inclk0 frequency: 50.000 MHz
Operation Mode: Normal

Clk	Ratio	Ph (dg)	Td (ns)	DC (%)
c0	1/1	0.00	0.00	50.00
e0	1/1	0.00	0.00	50.00

Compile the Design & Download to the Board

To compile the design and download it to the target board, perform the following steps:

1. Chose **Save** (File menu) to save changes to the BDF.

2. Choose **Start Compilation** (Processing menu) to start compiling the hardware design. The compilation begins.

If you performed all prior steps correctly, the Quartus II compilation will finish successfully after several minutes, and generate a new FPGA configuration file for the project.

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

To download the hardware design to the board, perform the following steps:

1. Connect your host computer to the development board via an Altera download cable, such as the USB Blaster.

2. Choose **Programmer** (Tools menu) to open the Quartus II Programmer.

3. Use the Programmer window to download the following FPGA configuration file to the board: **<*Hardware files directory*>\standard.sof**.

You have completed all the steps to create a multi-clock hardware design and download it to hardware. This design is based on the Nios II processor, and therefore you will have to run a software program on the processor to exercise the hardware.

Running Software to Exercise the Multi-Clock Hardware

In this section, you will run a program on the Nios II processor to exercise the multi-clock domain hardware. This program sets up and initiates a DMA transfer using the Nios II processor, and measures the time for the DMA transfer to complete.

☞ There is nothing special about this program that makes it specific to multi-clock domain systems. Because the Avalon switch fabric abstracts the details of clock domain crossing, the Nios II processor benefits from the fast performance of the DMA controller without needing to be aware of the system clock domain properties.

You will perform the following steps:

1. Install the example software design files.

2. Create a new Nios II IDE project using the software files.

3. Build and run the program.

4. Analyze the results.

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

Install the Example Software Design Files

In this section, you will install the example software design files on your computer. You can download the example design files from the Altera web site. Before you proceed installing the files, download the file **multi_clock.zip** associated with the URL for this chapter.

The file **multi_clock.zip** contains the following C-language source files:

- **dma_xfer.c** — Contains `main()`.
- **init.c**, **init.h** — Contain initialization routines to set up memory buffers, and initialize the timer.
- **settings_check.h** — Provides basic error checking to verify that the hardware contains the necessary DMA and memory components.

The file **multi_clock.zip** contains example software files packaged as a Nios II IDE software template. Perform the following steps to install the files:

1. Extract the contents of **multi_clock.zip** into a new directory called **multi_clock**.

2. Move the **multi_clock** directory under the following directory: <*Nios II EDS install path*>**\examples\software**

The files are packaged as a Nios II IDE template only to minimize the number of steps required for you to run the software. Using a template is not a necessary part of writing software for multi-clock domain systems. Using the template automatically provides the following functionality in the Nios II IDE, which you otherwise would have to perform manually:

1. Imports source files into a new project directory.

2. Sets up the system library settings.

3. Sets up the project settings.

Create a New Nios II IDE Project

To create a new Nios II IDE project using the provided example files, perform the following steps:

1. Start the Nios II IDE.

2. Choose **New > C/C++ Application** (File menu) to start a new project. The first page of the **New Project** wizard appears.

3. Under **Select Project Template**, select **Multi-Clock Domain Tutorial**.

4. Ensure that **Use Default Location** is turned on.

5. Click **Browse** under **Select Target Hardware**. The **Select Target Hardware** dialog box appears.

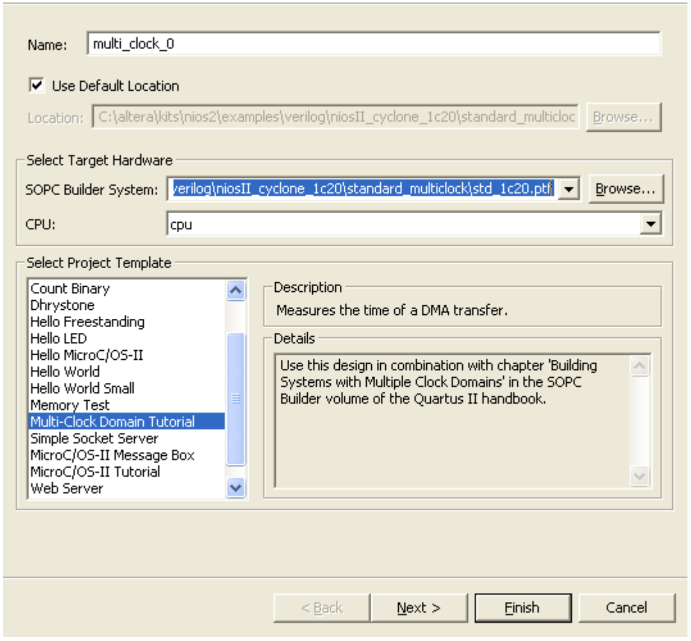
6. Browse to the <*hardware files directory*> directory where you created the hardware design earlier.

7. Select the file **std_<***FPGA***> .ptf**.

8. Click **Open** to return to the **New Project** wizard. The **SOPC Builder System** field is now specified and the **CPU** field now contains the name of the CPU in the system, as shown in Figure 11–9.

9. Click **Finish**.

Figure 11–9. New Project Wizard Filled in with Correct Settings


Name: multi_clock_0

☑ Use Default Location

Location: C:\altera\kits\nios2\examples\verilog\niosII_cyclone_1c20\standard_multicloc

Select Target Hardware

SOPC Builder System: verilog\niosII_cyclone_1c20\standard_multiclock\std_1c20.ptf

CPU: cpu

Select Project Template

Count Binary
Dhrystone
Hello Freestanding
Hello LED
Hello MicroC/OS-II
Hello World
Hello World Small
Memory Test
Multi-Clock Domain Tutorial
Simple Socket Server
MicroC/OS-II Message Box
MicroC/OS-II Tutorial
Web Server

Description

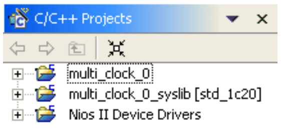
Measures the time of a DMA transfer.

Details

Use this design in combination with chapter 'Building Systems with Multiple Clock Domains' in the SOPC Builder volume of the Quartus II handbook.

After the IDE successfully creates the new project, the **C/C++ Projects** view contains two new projects, **multi_clock_0** and **multi_clock_0_syslib**, in addition to **Nios II Device Drivers**, as shown in Figure 11–10.

Figure 11–10. New Projects Displayed in the C/C++ Projects View



multi_clock_0
multi_clock_0_syslib [std_1c20]
Nios II Device Drivers

Build & Run the Program

In this section, you will build the software in the Nios II IDE and run it on a development board. In short, the program does the following:

- Initializes the read and write memory buffers, filling the read memory buffer with random values.
- Performs timer initialization to accurately measure how long a DMA transaction will take.
- Sets up a DMA transaction to copy the contents of the read buffer to the write buffer.
- Starts the timer, initiates the DMA transaction, and waits for the DMA to generate an interrupt.
- Stops the timer when the DMA transaction finishes.
- Verifies that the write buffer contents are correct.
- Reports the duration of the DMA transaction.

To build and run the program, perform the following steps:

1. In the Nios II IDE C/C++ Projects view, select the **multi_clock_0** project.

2. Choose **Run As** > **Nios II Hardware** (Run menu) to build the program, download it to the board, and run it. The IDE automatically builds the program before attempting to run it. The build process can take several minutes. After the build completes, the IDE will download the program to the target board and run it.

3. View the results in the **Console** view.

The **Console** view will display results similar to the following:

```
nios2-terminal: starting in terminal mode (Control-C
exits)

Hello from Nios II!
Starting DMA transfer...

Starting a DMA transfer of 4096 bytes of data.
DMA transfer completed.
It took 34.3600006104 useconds to complete the
transfer.
Comparing send and receive buffer data...
Data Matches.

Program completed successfully.
```

See the Nios II IDE online help for more information on building and downloading projects.

In this example, the DMA achieved approximately 120 MBytes per second (4096 bytes / 34.36 usec). The source (read) and destination (write) buffer memories have zero wait states, and therefore can perform a maximum of one transfer per clock cycle. For successive 32-bit transfers at 100 MHz, the theoretical maximum DMA transfer performance is 400 MBytes per second. The 34.36 usec time includes the following processor overhead, which accounts for the difference between 400 and 120:

- Time spent in the DMA driver setting up the DMA transfer.
- Time spent in the interrupt handler after the DMA flags that it has completed the transfer.
- Time spent entering the DMA callback function to capture the finish time.

Conclusion

Congratulations! You have completed all tutorial steps in this chapter to create a multi-clock domain system with SOPC Builder and exercise the system in hardware.