Introduction

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

This chapter is divided into the following sections:

SOPC Builder Components and the Component Editor

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

- HDL files—define the component's functionality as hardware.

- Hardware Component Description File (_hw.tcl) —describes the SOPC Builder related characteristics, such as interface behaviors. This file is created by the component editor.

- C-language files—define the component register map and driver software to allow programs to control the component.

- Software Component Description File (_sw.tcl) file—used by the software build tools to use and compile the component driver code.

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

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

Prerequisites

This chapter assumes that you are familiar with the following:

- Building systems with SOPC Builder. For details, refer to the Introduction to SOPC Builder chapter in volume 4 of the Quartus II Handbook.

- SOPC Builder components. For details, refer to the SOPC Builder Components chapter in volume 4 of the Quartus II Handbook.

- Basic concepts of the Avalon-MM interface.

Hardware and Software Requirements

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

■ Design files for the example design—A hyperlink to the design files appears next to the chapter, *SOPC Builder Component Development Walkthrough*, on the SOPC Builder literature page.

■ Nios development board and an Altera® USB-Blaster™ download cable—You can use either of the following Nios development boards:

■ Stratix® II Edition

■ Cyclone® II Edition

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

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

Component Development Flow

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

Typical Design Steps

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

1. Specification and definition.

   a. Define the functionality of the component.

   b. Determine component interfaces, such as Avalon Memory-Mapped (Avalon-MM), Avalon Streaming (Avalon-ST), interrupt, or other interfaces.

   c. Determine the component clocking requirements; what interfaces are synchronous to what clock inputs.

   d. If you want a microprocessor to control the component, determine the interface to software, such as the register map.

2. Implement the component in VHDL or Verilog HDL.

3. Import the component into SOPC Builder.

   a. Use the component editor to create a **_hw.tcl** file that describes the component.

   b. Instantiate the component into an SOPC Builder system.

   When importing an HDL file using the component editor, any parameter definitions that are dependent upon other defined parameters cause an error. Example 10–1 illustrates the declaration of a DEPTH parameter which is legal Verilog HDL syntax in the Quartus II software, but causes an error in the component editor syntax checker.

Example 10–1. DEPTH Parameter

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

To avoid this error, use a localparam for the dependent parameter instead, as shown in Example 10–2.

Example 10–2. localparam Parameter

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

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

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

5. Perform in-system testing, such as the following:

   a. Test register-level accesses to the component in hardware or simulation using a microprocessor, such as the Nios II processor.

   b. Performance benchmarking.

Hardware Design

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

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

■ Task logic—Implements the component's fundamental function. The task logic is design dependent.

■ Interface logic—Provides a standard way of providing data to or getting data from the components and of controlling the functioning of the components.

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

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

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

Design Example: Checksum Hardware Accelerator

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

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

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

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

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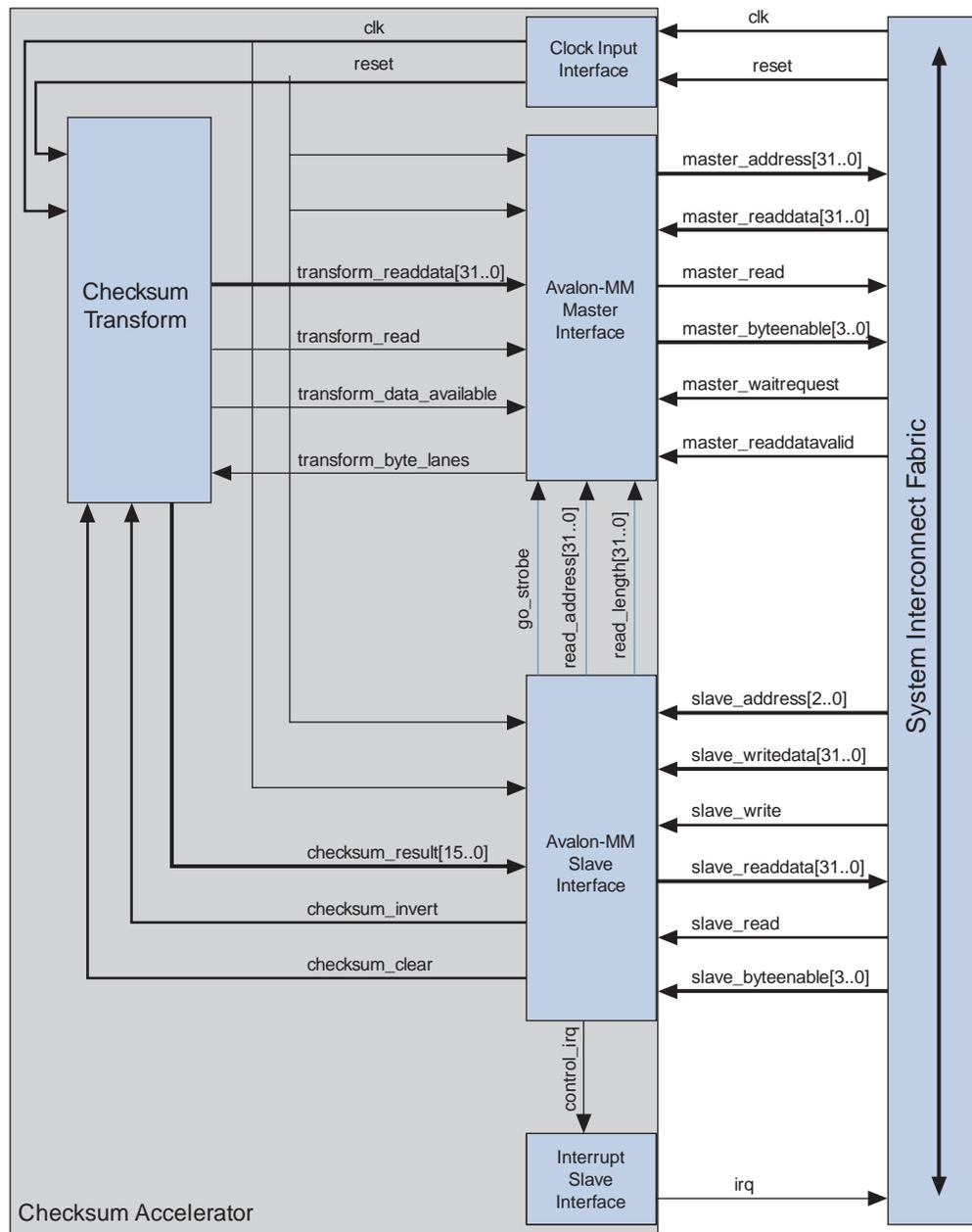
Software Design

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

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

Table 10–1. Avalon-MM Slave Port Register Map (Control)

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

Note to Table 10–1:

(1) Wr=Writable; Rd=Readable; Wclr=Write 1 to clear

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

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

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

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

Verifying the Component

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

System Console

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

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

System-Level Verification

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

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

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

Sharing Components

When you create a component, component editor saves the **_hw.tcl** file in the same directory as the top-level HDL file. Where appropriate, files referenced by the **_hw.tcl** file are specified relative to the **_hw.tcl** file itself, so the files can easily be moved and copied. To share a component, include it in your IP library.

For more information about including components in an IP library refer to *Finding Components in SOPC Builder* in *Chapter 4: SOPC Builder Components* in volume 4 of the *Quartus II Handbook.*

.sopcinfo Files

Every time SOPC Builder generates a system, a *<mysystem>*.**sopcinfo** file is also generated, which contains the information described below.

- SOPC Builder project, including:

    - Name and tool version

    - HDL language

- Each module instantiated in the system, including:

    - Name and version

    - Where interface information was found on the disk, such as signal names and types, interface properties, and clock domain mapping

    - Parameter names and values

■ Each connection, including:

■ Component and interface connections

■ Base address, Avalon-MM interfaces, IRQ number interfaces

■ Memory map as seen by each master in the system

☞ The **.sopcinfo** file is a report file only, and cannot be edited with SOPC Builder.

Referenced Documents

This chapter references the following documents:

■ *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*

■ *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*

■ *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*

■ *System Console User Guide*

■ *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*

Document Revision History

Table 10–2 shows the revision history for this chapter.

Table 10–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2008, v8.1.0	■ Added reference to new search path for IP chapter 4 of this volume. ■ Correction direction of signals in Figure 10–1. ■ Changed page size to 8.5 x 11 inches.	One correction and one change to reflect changes in underlying software.
May 2008, v8.0.0	■ Chapter renumbered from 9 to 10. ■ Removed discussion of the Checksum Design example, which will now be in a readme.pdf file and zipped with the rest of the design files. ■ Deleted references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications. ■ New Figure 9-1 and Table 9-1. ■ New section on .**sopcinfo** file.	Deleted example procedure.
October 2007, v7.2.0	Updated instructions on how to develop components to match new GUI.	—
May 2007, v7.1.0	Changed example component from a pulse width modulator with that only has an Avalon-MM slave interface to a checksum master that includes both Avalon-MM master and slave interfaces.	Changed the example design to one with more practical applications. Updated instructions for the 7.1 release.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	Chapter 9 was previously chapter 10. No change to content.	—
May 2006, v6.0.0	Chapter 10 was previously chapter 9. No change to content.	—
October 2005, v5.1.0	Chapter 9 was previously chapter 7. No change to content.	—
August 2005, v5.0.1	Corrected Table 7-5.	—
May 2005, v5.0.0	No change from previous release.	—
February 2005, v1.0	Initial release.	—