Embedded Design Handbook

ED_HANDBOOK
2016.12.19

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

Contents

Introduction 1

2016.12.19

ED_HANDBOOK Subscribe Send Feedback

The Embedded Design Handbook complements the primary documentation for the Altera® tools for embedded system development. It describes how to most effectively use the tools, and recommends design styles and practices for developing, debugging, and optimizing embedded systems using Altera-provided tools. The handbook introduces concepts to new users of Altera's embedded solutions, and helps to increase the design efficiency of the experienced user.

Document Revision History

Table 1-1: Introduction Chapter Revision History

Date	Version	Changes
December 2016	2016.12.19	Maintenance Release.
December 2015	2015.12.18	Initial release.

First Time Designer's Guide 2

2016.12.19

ED_HANDBOOK Subscribe Send Feedback

First Time Designer's Guide is a basic overview of Altera embedded development process and tools for the first time user. The chapter provides information about the design flow and development tools' interactions, and describes the differences between the Nios II processor flow and a typical discrete microcontroller design flow.

This guide does not replace the basic reference material for the first time designer. It references other documents that provide detailed information about the individual tools and procedures. It contains resources and sections to help the first-time user of Altera's embedded development tools for hardware and software development. For more information, refer to the related information links.

Related Information

- **Nios II Classic Processor Reference Guide**
- **Nios II Gen2 Processor Reference Guide**
- **Nios II Classic Software Developer's Handbook**
- **Nios II Gen2 Software Developer's Handbook**
- **Embedded Peripherals IP User Guide**
- **Qsys System Development**
- **Nios II Flash Programmer User Guide**

FPGAs and Soft-Core Processors

FPGAs can implement logic that functions as a complete microprocessor while providing many flexibility options.

An important difference between discrete microprocessors and FPGAs is that an FPGA contains no logic when it powers up. Before you run software on a Nios II based system, you must configure the FPGA with a hardware design that contains a Nios II processor. To configure an FPGA is to electronically program the FPGA with a specific logic design. The Nios II processor is a true soft-core processor: it can be placed anywhere on the FPGA, depending on the other requirements of the design. Two different sizes of the processor are available for Nios II Gen2, each with flexible features.(1)

(1) There are three different sizes of the processor that are available for Nios II Classic.

To enable your FPGA-based embedded system to behave as a discrete microprocessor-based system, your system should include the following:

- A JTAG interface to support FPGA configuration and hardware and software debugging
- A power-up FPGA configuration mechanism

If your system has these capabilities, you can begin refining your design from a pretested hardware design loaded in the FPGA. Using an FPGA also allows you to modify your design quickly to address problems or to add new functionality. You can test these new hardware designs easily by reconfiguring the FPGA using your system's JTAG interface.

The JTAG interface supports hardware and software development. You can perform the following tasks using the JTAG interface:

- Configure the FPGA
- Download and debug software
- Communicate with the FPGA through a UART-like interface (JTAG UART)
- Debug hardware (with the SignalTap® II embedded logic analyzer)
- Program flash memory

After you configure the FPGA with your Nios II processor-based design, the software development flow is similar to the flow for discrete microcontroller designs.

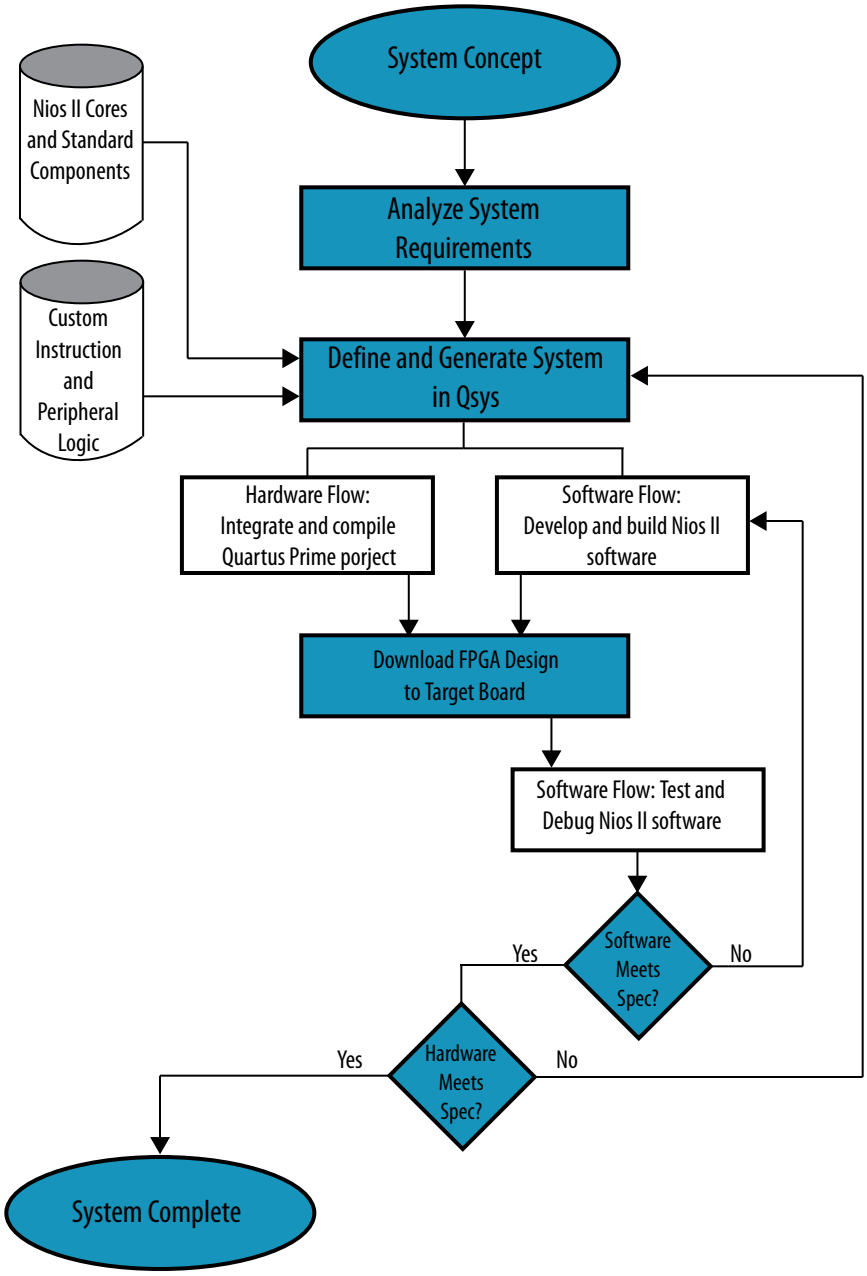
Embedded System Design

Whether you are a hardware designer or a software designer, read the *Nios II Hardware Development Tutorial* to start learning about designing embedded systems on an Altera FPGA. The "Nios II System Development Flow" section is particularly useful in helping you to decide how to approach system design using Altera's embedded hardware and software development tools. Altera recommends that you read this tutorial before starting your first design project. The tutorial teaches you the basic hardware and software flow for developing Nios II processor-based systems.

Designing with FPGAs gives you the flexibility to implement some functionality in discrete system components, some in software, and some in FPGA-based hardware. This flexibility makes the design process more complex. The Qsys system design tool helps to manage this complexity. Even if you decide a soft-core processor does not meet your application's needs, Qsys can still play a vital role in your system by providing mechanisms for peripheral expansion or processor offload.

The figure below illustrates the overall Nios II system design flow, including both hardware and software development. This illustration is greatly simplified. There are numerous correct ways to use the Altera tools to create a Nios II system.

Figure 2-1: General Nios II System Design Flow



Related Information

- **Hardware System Design with Quartus Prime and Qsys** on page 3-1
- **Software System Design with a Nios II Processor** on page 4-1

Embedded Design Resources

This section contains a list of resources to help you find design help. Your resource options include traditional Altera-based support such as online documentation, training, and My Support, as well as web-based forums and Wikis. The best option depends on your inquiry and your current stage in the design cycle.

Altera Embedded Support

Altera recommends that you seek support in the following order:

1. Look for relevant literature on the Altera Documentation page, especially on the Documentation: Nios II Processor page.
2. Contact your local Altera sales office or sales representative, or your field application engineer (FAE).
3. Contact technical support through the myAltera page of the Altera website to get support directly from Altera.
4. Consult one of the following community-owned resources:
   - The Nios Forum, available on the Altera Forum website
   - The Altera Wiki website
   - Rocketboards for Linux support

   Note: Altera is not responsible for the contents of the Nios Forum and Altera Wiki websites, which are maintained by public authors and experts outside of Altera.

Related Information

- **Altera Documentation**
- **myAltera**
- **Altera Forum**
- **Altera Wiki**
- **Documentation: Nios II Processor**
- **Rocketboards**

Altera Embedded Training

To learn how the tools work together and how to use them in an instructor-led environment, register for training. Several training options are available. For information about general training, refer to the Training page of the Altera website.

For detailed information about available courses and their locations, visit the Embedded SW Designer Curriculum page of the Altera website. This page contains information about both online and instructor-led training.

Related Information

- **Altera Training**
- **Embedded SW Designer Curriculum**

Altera Embedded Documentation

You can access documentation about the Nios II processor and embedded design from your Nios II EDS installation directory at <*Nios II EDS install dir*>**\documents\index.htm**. To access this page directly on

Windows platforms, on the Start menu, click **All Programs**. On the All Programs menu, on the Altera submenu, on the Nios II EDS <version> submenu, click **Nios II**<*version*>**Documentation**. This web page contains links to the latest Nios II documentation.

The Documentation: Nios II Processor page of the Altera website includes a list and links to available documentation. At the bottom of this page, you can find links to various product pages that include Nios II processor online demonstrations and embedded design information.

The other chapters in the *Embedded Design Handbook* are a valuable source of information about embedded hardware and software design, verification, and debugging. Each chapter contains links to the relevant overview documentation.

Related Information

Documentation: Nios II Processor

Third Party Intellectual Property

Many third parties have participated in developing solutions for embedded designs with Altera FPGAs through the Altera AMPPSM Program. For up-to-date information about the third-party solutions available for the Nios II processor, visit the Nios II Processor page of the Altera website, and select the **Ecosystem** tab.

Several community forums are also available. These forums are not controlled by Altera. The Altera Forum's Marketplace provides third-party hard and soft embedded systems-related IP. The forum also includes an unsupported projects repository of useful example designs. You are welcome to contribute to these forum pages.

Traditional support is available from the Support Center or through your local Field Application Engineer (FAE). You can obtain more informal support by visiting the Nios Forum section of the Altera Forum or by browsing the information contained on the Altera Wiki. Many experienced developers, from Altera and elsewhere, contribute regularly to Wiki content and answer questions on the Nios Forum.

Related Information

- **Embedded Processing Page**
- **www.alteraforum.com**
- **www.alterawiki.com**

Altera Embedded Glossary

The following definitions explain some of the unique terminology for describing Qsys and Nios II processor-based systems:

- **Component**—A named module in Qsys that contains the hardware and software necessary to access a corresponding hardware peripheral.
- **Custom instruction**—Custom hardware processing integrated with the Nios II processor's ALU. The programmable nature of the Nios II processor and Qsys-based design supports this implementation of software algorithms in custom hardware. Custom instructions accelerate common operations. (The Nios II processor floating-point instructions are implemented as custom instructions).
- **Custom peripheral**—An accelerator implemented in hardware. Unlike custom instructions, custom peripherals are not connected to the CPU's ALU. They are accessed through the system interconnect fabric. (See System interconnect fabric). Custom peripherals offload data transfer operations from the processor in data streaming applications.
- **ELF (Executable and Linking Format)**—The executable format used by the Nios II processor. This format is arguably the most common of the available executable formats. It is used in most of today's popular Linux/BSD operating systems.
- **HAL (Hardware Abstraction Layer)**—A lightweight runtime environment that provides a simple device driver interface for programs to communicate with the underlying hardware. It provides a POSIX-like software layer and wrapper to the newlib C library.
- **Nios II Command Shell**—The command shell you use to access Nios II and Qsys command-line utilities.
  - On Windows platforms, a Nios II Command Shell is a Cygwin bash with the environment properly configured to access command-line utilities.
  - On Linux platforms, to run a properly configured bash, type `<Nios II EDS install path>/nios2_command_shell.sh`
- **Nios II Embedded Development Suite (EDS)**—The complete software environment required to build and debug software applications for the Nios II processor.
- **Nios II Software Build Tools (SBT)**—Software that allows you to create Nios II software projects, with detailed control over the software build process.
- **Nios II Software Build Tools for Eclipse**—An Eclipse-based development environment for Nios II embedded designs, using the SBT for project creation and detailed control over the software build process. The SBT for Eclipse provides software project management, build, and debugging capabilities.
- **Qsys**—Software that provides a GUI-based system builder and related build tools for the creation of FPGA-based subsystems, with or without a processor.
- **System interconnect fabric**—An interface through which the Nios II processor communicates to on- and off-chip peripherals. This fabric provides many convenience and performance-enhancing features.

Document Revision History

Table 2-1: First Time Designer's Guide Chapter Revision History

Date	Version	Changes
December 2016	2016.12.19	Updates: • First Time Designer's Guide section updated • Nios II Software Design section moved to *Software System Design with a Nios II Processor* chapter
December 2015	2015.12.18	Updates: • Removed mention of SOPC Builder, now Qsys • Removed mention of C2H Complier • Quartus II now Quartus Prime • Removed mention of FS2 Console • Removed mention of Nios II IDE Sections removed: • Nios II IDE Flow
July 2011	2.3	• Clarified this handbook does not include information about Qsys. • Updated location of hardware design examples. • Updated references.
March 2010	2.2	Updated for the SBT for Eclipse.
January 2009	2.1	Updated Nios Wiki hyperlink.
Novemeber 2008	2.0	Added System Console.
March 2008	1.0	Initial release.

Hardware System Design with Quartus Prime and Qsys

3

2016.12.19

ED_HANDBOOK

This chapter provides information on the hardware system design flow, designing with Qsys, and interfacing an external processor to an FPGA. Also included are useful configurations available with a Nios II processor, timing constraints and requirements, and how to customize the FPGA to your design needs.

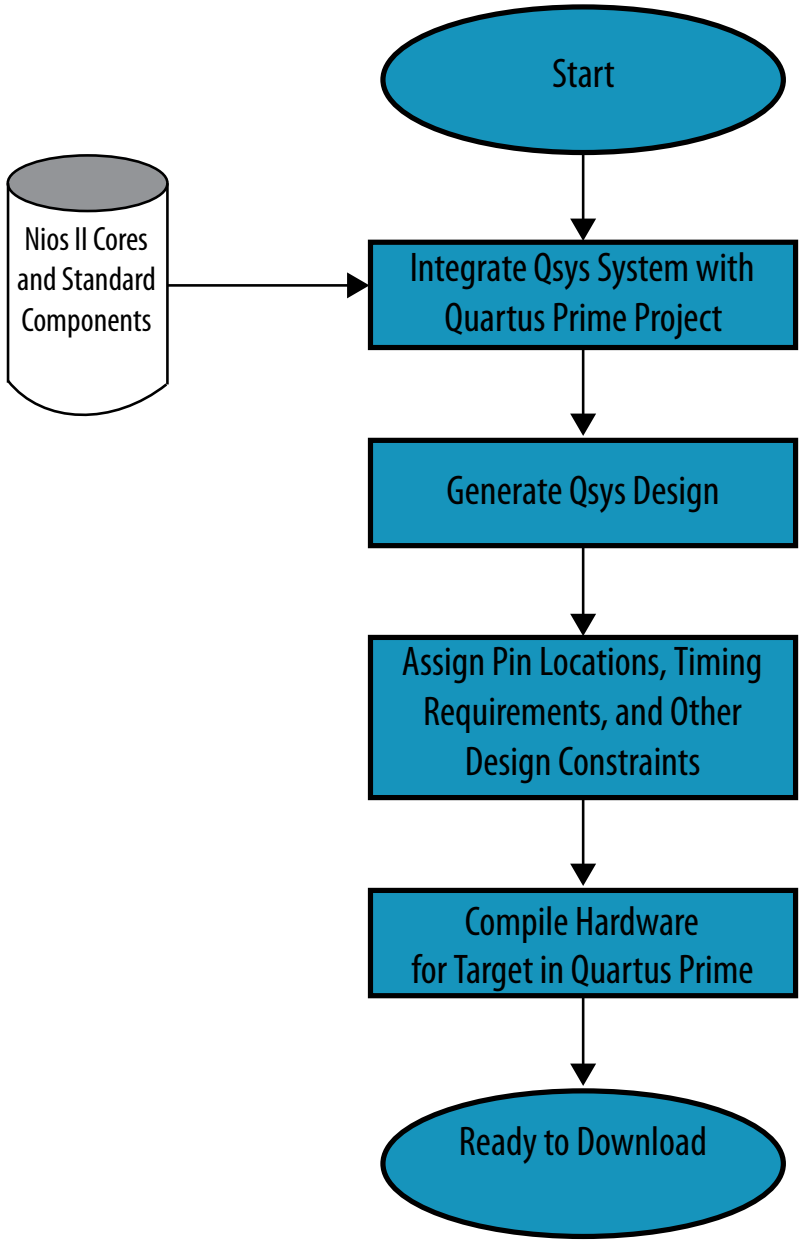
The Qsys system integration tool saves significant time and effort in the FPGA design process by automatically generating interconnect logic to connect intellectual property (IP) functions and subsystems. The following sections will explain how to connect signals in Qsys.

FPGA Hardware Design

Although you develop your FPGA-based design in Qsys, you must perform the following tasks in other tools:

- Connect signals from your FPGA-based design to your board level design
- Connect signals from your Qsys system to other signals in the FPGA logic
- Constrain your design

Figure 3-1: Nios II System Hardware Design Flow



Connecting Your FPGA Design to Your Hardware

To connect your FPGA-based design to your board-level design, perform the following two tasks:

- Identify the top level of your FPGA design.
- Assign signals in the top level of your FPGA design to pins on your FPGA using any of the methods mentioned at the Altera I/O Management, Board Development Support, and Signal Integrity Analysis Resource Center page of the Altera website.

Note: The top level of your FPGA-based design might be your Qsys system. However, the FPGA can include additional design logic.

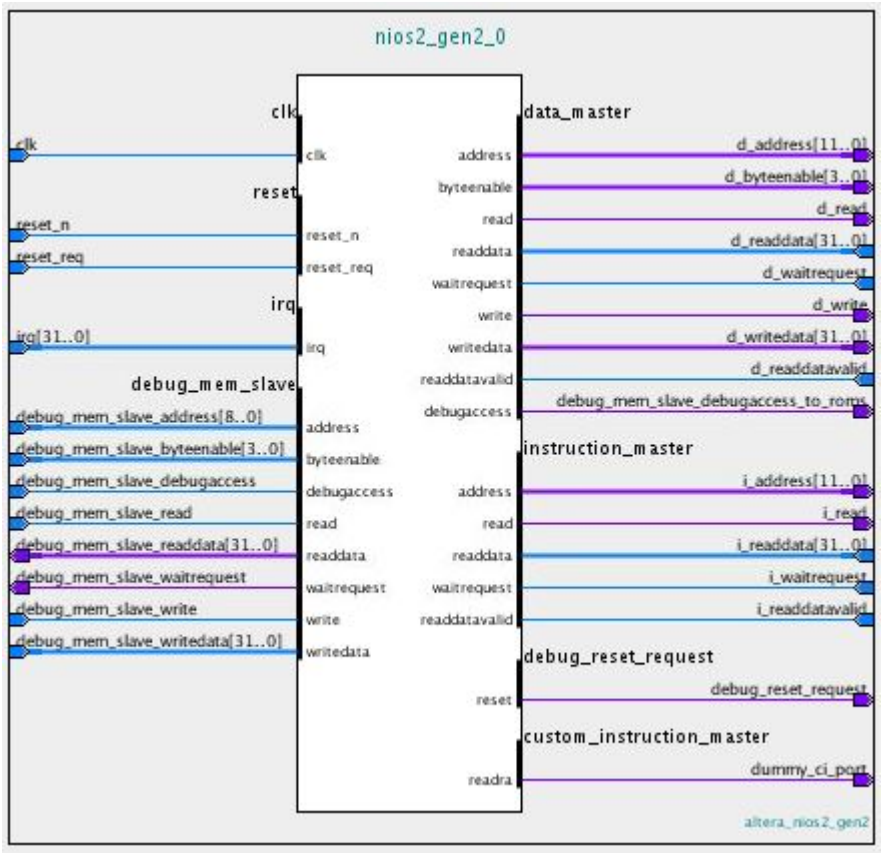
Related Information

I/O Management, Board Development Support, and Signal Integrity Analysis Resource Center

Connecting Signals to your Qsys System

You must define the clock and reset pins for your Qsys system. You must also define each I/O signal that is required for proper system operation. The figure below shows the top-level block diagram of a Qsys system that includes a Nios II processor. The large symbol in this top-level diagram, labeled `std_1s40`, represents the Qsys system. The flag-shaped pin symbols in this diagram represent off-chip (off-FPGA) connections.

Figure 3-2: Top-level Block Diagram



For more information about connecting your FPGA pins, refer to the Altera I/O Management, Board Development Support, and Signal Integrity Analysis Resource Center page of the Altera website.

Related Information

I/O Management, Board Development Support, and Signal Integrity Analysis Resource Center

Constraining Your FPGA-Based Design

To ensure your design meets timing and other requirements, you must constrain the design to meet these requirements explicitly using tools provided in the Quartus® Prime software or by a third party EDA provider. The Quartus Prime software uses your constraint information during design compilation to achieve Altera's best possible results.

Note: Altera's third-party EDA partners and the tools they provide are listed on Altera's Partner Solutions page of the Altera website.

Related Information

Altera's Partner Solutions

Hardware Design with Qsys

Qsys simplifies the task of building complex hardware systems on an FPGA. Qsys allows you to describe the topology of your system using a graphical user interface (GUI) and then generate the hardware description language (HDL) files for that system. The Quartus Prime software compiles the HDL files to create an SRAM Object File (**.sof**). For additional information about Qsys, refer to the Quartus Prime Handbook.

Qsys allows you to choose the processor core type and the level of cache, debugging, and custom functionality for each Nios II processor. Your design can use on-chip resources such as memory, PLLs, DSP functions, and high-speed transceivers. You can construct the optimal processor for your design using Qsys.

After you construct your system using Qsys, and after you add any required custom logic to complete your top-level design, you must create pin assignments using the Quartus Prime software. The FPGA's external pins have flexible functionality, and a range of pins is available to connect to clocks, control signals, and I/O signals.

For information about how to create pin assignments, refer to Quartus Prime Help and to the I/O Management chapter in Volume 2: Design Implementation and Optimization of the *Quartus Prime Handbook*.

Altera recommends that you start your design from a small pretested project and build it incrementally. Start with one of the many Qsys example designs available from the All Design Examples web page of the Altera website, or with an example design from the *Nios II Hardware Development Tutorial*.

Qsys allows you to create your own custom components using the component editor. In the component editor you can import your own source files, assign signals to various interfaces, and set various component and parameter properties.

Before designing a custom component, you should become familiar with the interface and signal types that are available in Qsys.

Native addressing is deprecated. Therefore, you should use dynamic addressing for slave interfaces on all new components. Dynamically addressable slave ports include byte enables to qualify which byte lanes are accessed during read and write cycles. Dynamically addressable slave interfaces have the added benefit of being accessible by masters of any data width without data truncation or side effects.

To learn about the interface and signal types that you can use in Qsys, refer to *Avalon Interface Specifications*. To learn about using the component editor, refer to the Component Editor chapter in the *Quartus Prime Handbook*.

As you add each hardware component to the system, test it with software. If you do not know how to develop software to test new hardware components, Altera recommends that you work with a software engineer to test the components.

The Nios II EDS includes several software examples, located in your Nios II EDS installation directory (**nios2eds**), at `<Nios II EDS install dir>\examples\software`. After you run a simple software design—such as the simplest example, Hello World Small—build individual systems based on this design to test the additional interfaces or custom options that your system requires. Altera recommends that you start with a simple system that includes a processor with a JTAG debug module, an on-chip memory component, and a JTAG UART component, and create a new system for each new untested component, rather than adding in new untested components incrementally.

After you verify that each new hardware component functions correctly in its own separate system, you can combine the new components incrementally in a single Qsys system. Qsys supports this design methodology well, by allowing you to add components and regenerate the project easily.

For detailed information about how to implement the recommended incremental design process, refer to the Verification and Board Bring-Up chapter of the *Embedded Design Handbook*.

Related Information

- **Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis**
- **All Design Examples**
- **Nios II Hardware Development Tutorial**
- **Avalon Interface Specifications**
- **Verification and Board Bring-Up** on page 6-18

Altera System on a Programmable Chip (Qsys) Solutions

To understand the Nios II software development process, you must understand the definition of a Qsys system. Qsys is a system development tool for creating systems including processors, peripherals, and memories. The tool enables you to define and generate a complete Qsys very efficiently. Qsys does not require that your system contain a Nios II processor, although it provides complete support for integrating Nios II processors with your system.

An Qsys system is similar in many ways to a conventional embedded system; however, the two kinds of system are not identical. An in-depth understanding of the differences increases your efficiency when designing your Qsys system.

In Altera Qsys solutions, the hardware design is implemented in an FPGA device. An FPGA device is volatile—contents are lost when the power is turned off— and reprogrammable. When an FPGA is programmed, the logic cells inside it are configured and connected to create a Qsys system, which can contain Nios II processors, memories, peripherals, and other structures. The system components are connected with Avalon® interfaces. After the FPGA is programmed to implement a Nios II processor, you can download, run, and debug your system software on the system.

Understanding the following basic characteristics of FPGAs and Nios II processors is critical for developing your Nios II software application efficiently:

- FPGA devices and Qsys—basic properties:
  - **Volatility**—The FPGA is functional only after it is configured, and it can be reconfigured at any time.
  - **Design**—Many Altera Qsys systems are designed using Qsys and the Quartus Prime software, and may include multiple peripherals and processors.
  - **Configuration**—FPGA configuration can be performed through a programming cable, such as the USB-Blaster™ cable, which is also used for Nios II software debugging operations.
  - **Peripherals**—Peripherals are created from FPGA resources and can appear anywhere in the Avalon memory space. Most of these peripherals are internally parameterizable.
- Nios II processor—basic properties:
  - **Volatility**—The Nios II processor is volatile and is only present after the FPGA is configured. It must be implemented in the FPGA as a system component, and, like the other system components, it does not exist in the FPGA unless it is implemented explicitly.
  - **Parameterization**—Many properties of the Nios II processor are parameterizable in Qsys, including core type, cache memory support, and custom instructions, among others.
  - **Processor Memory**—The Nios II processor must boot from and run code loaded in an internal or external memory device.
  - **Debug support**—To enable software debug support, you must configure the Nios II processor with a debug core. Debug communication is performed through a programming cable, such as the USB-Blaster cable.
  - **Reset vector**—The reset vector address can be configured to any memory location.
  - **Exception vector**—The exception vector address can be configured to any memory location.

Qsys Design

The recommended design flow requires that you maintain several small Qsys systems, each with its Quartus Prime project and the software you use to test the new hardware. A Qsys design requires the following files and folders:

- Quartus Prime Project File (**.qpf**)
- Quartus Prime Settings File (**.qsf**)

  The **.qsf** file contains all of the device, pin, timing, and compilation settings for the Quartus Prime project.
- One of the following types of top-level design file:
  - Block Design File (**.bdf**)
  - Verilog Design File (**.v**)
  - VHDL Design File (**.vhd**)

    If Qsys generates your top-level design file, you do not need to preserve a separate top-level file.

    Qsys generates most of the HDL files for your system, so you do not need to maintain them when preserving a project. You need only preserve the HDL files that you add to the design directly.

    For details about the design file types, refer to the Quartus Prime Help.
- Qsys Design File (**.sopc**)
- Qsys Information File (**.sopcinfo**)

  This file contains an XML description of your Qsys system. Qsys and downstream tools, including the Nios II Software Build Tools (SBT), derive information about your system from this file.
- Your software application source files

To replicate an entire project (both hardware and software), simply copy the required files to a separate directory. You can create a script to automate the copying process. After the files are copied, you can proceed to modify the new project in the appropriate tools: the Quartus Prime software, Qsys, the SBT for Eclipse, the SBT in the command shell, or the Nios II Integrated Development Environment (IDE).

For more information about all of these files, refer to the "Archiving Projects" chapter in the *Quartus Prime Handbook Volume 1: Design and Synthesis*.

Related Information

Archiving Projects

Interfacing an External Processor to an Altera FPGA

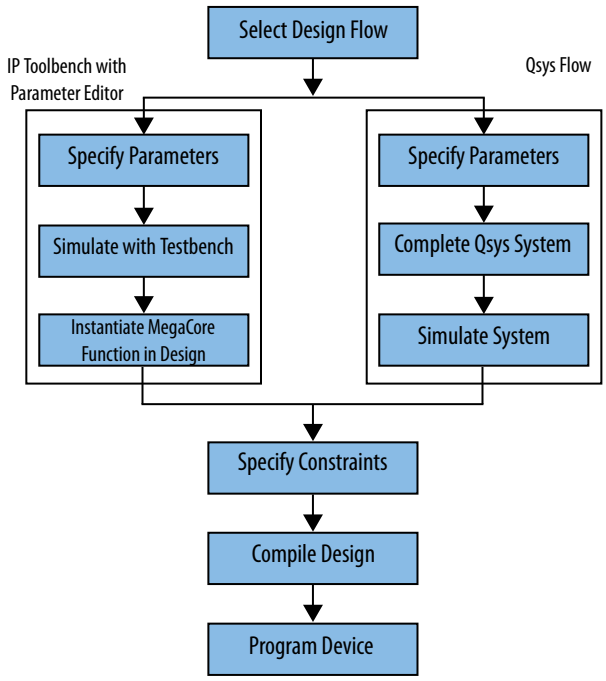
Altera provides options to connect an external processor to an Altera FPGA. These interface options include the PCI Express, PCI, RapidIO™, serial peripheral interface (SPI) interface or a simple custom bridge that you can design yourself.

By including both an FPGA and a commercially available processor in your system, you can partition your design to optimize performance and cost in the following ways:

- Offload pre- or post- processing of data to the external processor
- Create dedicated FPGA resources for co-processing data
- Reduce design time by using IP from Altera's library of components to implement peripheral expansion for industry standard functionality
- Expand the I/O capability of your external processor

You can instantiate the PCI Express, PCI, and RapidIO MegaCore functions using either the Parameter Editor or Qsys design flow. The PCI Lite and SPI cores are only available in the Qsys design flow. Qsys automatically generates the HDL design files that include all of the specified components and system connectivity. Alternatively, you can use the IP Toolbench with the Parameter Editor to generate a stand-alone component outside of Qsys. The figure below shows the steps you take to instantiate a component in both design flows.

Figure 3-3: Qsys and Parameter Editor Design Flows



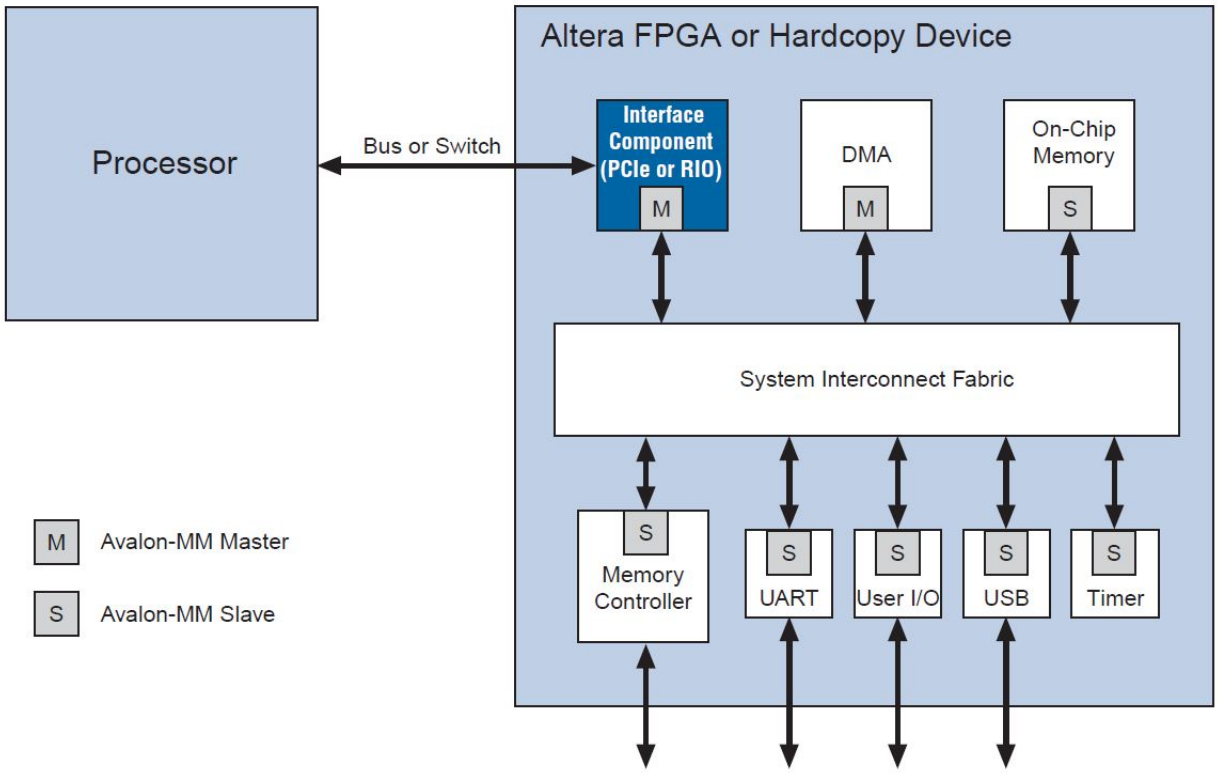
The remainder of this section provides an overview of the MegaCore functions that you can use to interface an Altera FPGA to an external processor. It covers the following topics:

- Configuration Options
- RapidIO Interface
- PCI Express Interface
- PCI Interface
- Serial Protocol Interface (SPI)
- Custom Bridge Interfaces

Configuration Options

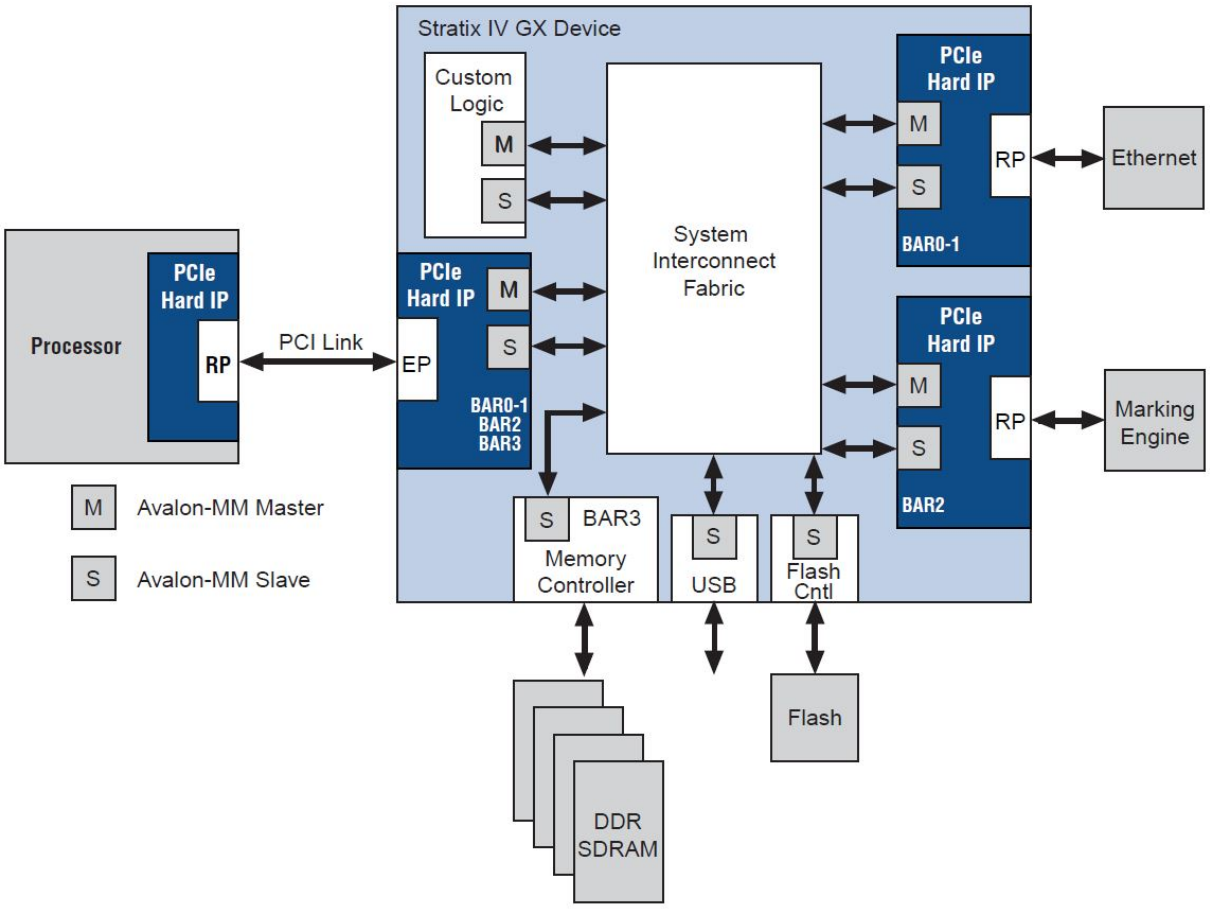
The figure below illustrates a Qsys system design that includes a high-performance external bus or switch to connect an industry-standard processor to an external interface of a MegaCore function inside the FPGA. This MegaCore function also includes an Avalon-MM master port that connects to the Qsys system interconnect fabric. As the figure illustrates, Altera provides a library of components, typically Avalon-MM slave devices, that connect seamlessly to the Avalon system interconnect fabric.

Figure 3-4: FPGA with a Bus or Switch Interface Bridge for Peripheral Expansion



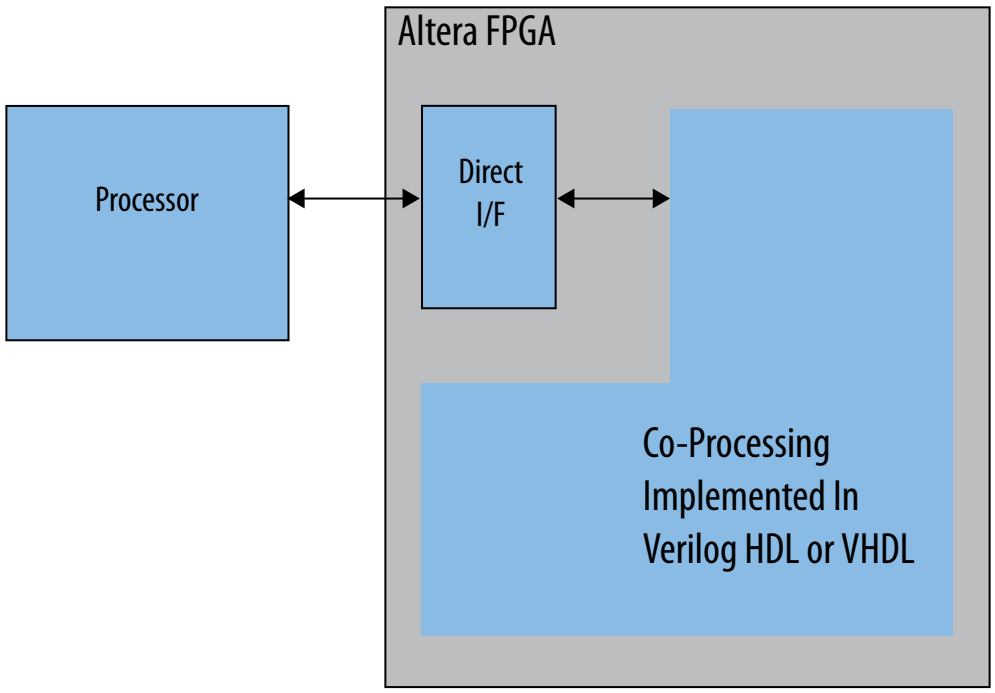
The design below includes an external processor that interfaces to a PCI Express endpoint inside the FPGA. The system interconnect fabric inside the implements a partial crossbar switch between the endpoint that connects to the external processor and two additional PCI Express root ports that interface to an Ethernet card and a marking engine. In addition, the system includes some custom logic, a memory controller to interface to external DDR SDRAM memory, a USB interface port, and an interface to external flash memory. Qsys automatically generates the system interconnect fabric to connect the components in the system.

Figure 3-5: FPGA with a Processor Bus or SPI for Peripheral Expansion



Alternatively, you can also implement your logic in Verilog HDL or VHDL without using Qsys. Below the figure illustrates a modular design that uses the FPGA for co-processing with a second module to implement the interface to the processor. If you choose this option, you must write all of the HDL to connect the modules in your system.

Figure 3-6: FPGA Performs Co-Processing



The table below summarizes the components Altera provides to connect an Altera FPGA device to an external processor. As this table indicates, three of the components are also available for use in the Parameter Editor design flow in addition to Qsys. Alternative implementations of these components are also available through the Altera IP Core Partners Program (AMPPSM) partners. The AMPP partners offer a broad portfolio of IP cores optimized for Altera devices.

For a complete list of third-party IP for Altera FPGAs, refer to the Intellectual Property: Reference Designs web page of the Altera website. For Qsys components, search for `sopc_builder_ready` in the IP MegaStore IP core search function.

Table 3-1: Processor Interface Solutions Available from an Altera Device

Protocol	Available in Qsys	Available In Parameter Editor	Third-Party Solution	OpenCore Plus Evaluation Available
RapidIO	Yes	Yes	Yes	Yes
PCI Express	Yes	Yes	Yes	Yes
PCI	Yes	Yes	Yes	Yes
PCI Lite	Yes	—	—	License not required
SPI	Yes	—	—	

The table below summarizes the most popular options for peripheral expansion in Qsys systems that include an industry-standard processor. All of these are available in Qsys. Some are also available using the Parameter Editor.

Table 3-2: Partial list of peripheral interfaces available for Qsys

Protocol	Available in Qsys	Available In Parameter Editor	Third-Party Solution	OpenCore Plus Evaluation Available
CAN	Yes	—	Yes	Yes
I2C	Yes	—	Yes	Yes
Ethernet	Yes	Yes	Yes	Yes
PIO	Yes	—	—	Not required
POS-PHY Level 4 (SPI 4.2)	—	Yes	—	Yes
SPI	Yes	—	Yes	Not required
UART	Yes	—	Yes	Yes
USB	Yes	—	Yes	Yes

For detailed information about the components available in Qsys refer to the *Embedded Peripherals IP User Guide* and the **Intellectual Property: Find IP** page.

In some cases, you must download third-party IP solutions from the AMPP vendor website, before you can evaluate the peripheral using the OpenCore Plus.

For more information about the AMPP program and OpenCore Plus refer to *AN343: OpenCore Evaluation of AMPP Megafunctions* and *AN320: OpenCore Plus Evaluation of Megafunctions*.

The following sections discuss the high-performance interfaces that you can use to interface to an external processor.

Related Information

- **Intellectual Property: Reference Designs**
- **Embedded Peripherals IP User Guide**
- **OpenCore Evaluation of AMPP Megafunctions**
- **OpenCore Plus Evaluation of Megafunctions**
- **Intellectual Property: Find IP**

RapidIO Interface

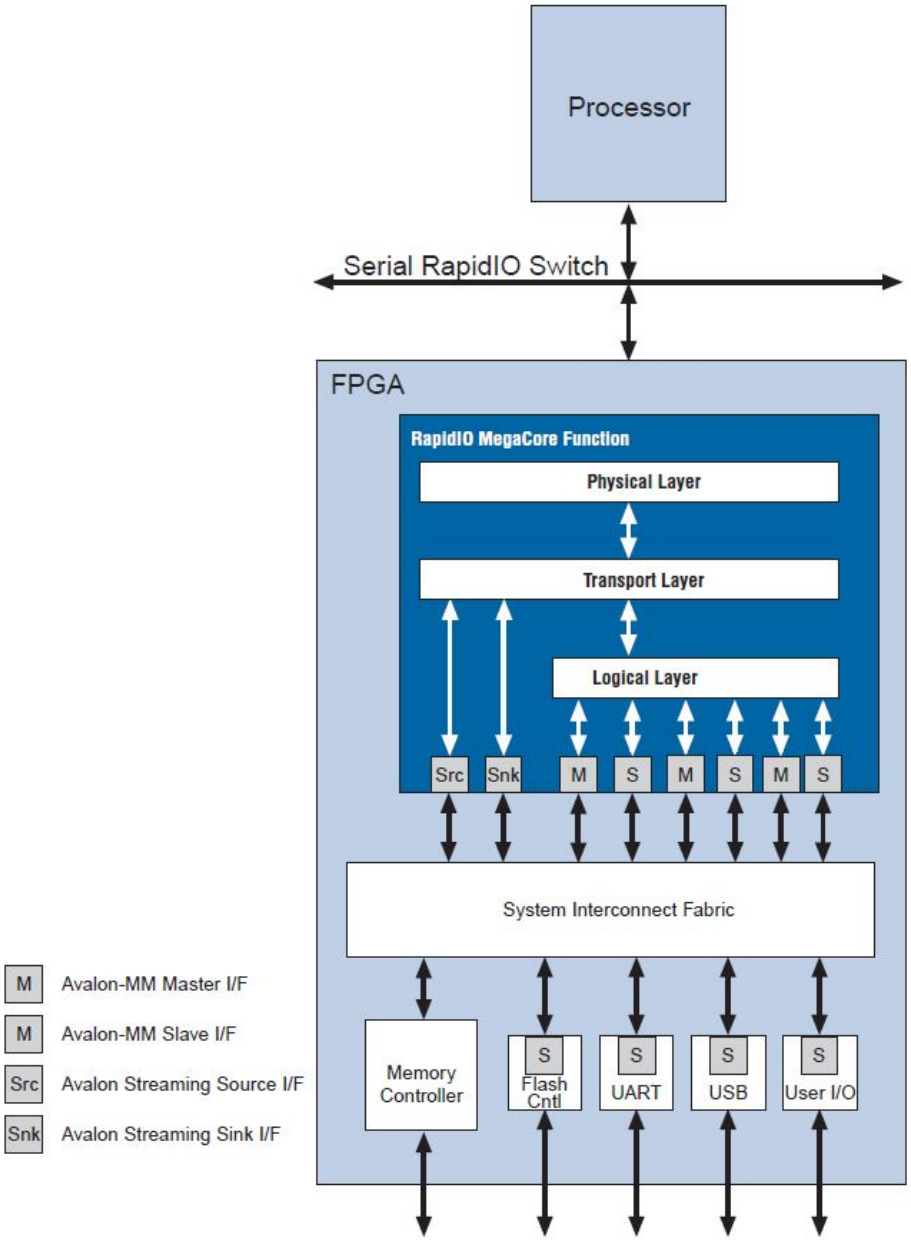
RapidIO is a high-performance packet-switched protocol that transports data and control information between processors, memories, and peripheral devices. The RapidIO MegaCore function is available in Qsys includes Avalon-MM ports that translate Serial RapidIO transactions into Avalon-MM transactions. The MegaCore function also includes an optional Avalon Streaming (Avalon-ST) interface that you can use to send transactions directly from the transport layer to the system interconnect fabric. When you select all optional features, the core includes the following ports:

- Avalon-MM I/O write master
- Avalon-MM I/O read master
- Avalon-MM I/O write slave
- Avalon-MM I/O read slave
- Avalon-MM maintenance master

- Avalon-MM system maintenance slave
- Avalon Streaming sink pass-through TX
- Avalon-ST source pass-through RX

Using the Qsys design flow, you can integrate a RapidIO endpoint in a Qsys system. You connect the ports using the Qsys **System Contents** tab and Qsys automatically generates the system interconnect fabric. The figure below illustrates a Qsys system that includes a processor and a RapidIO MegaCore function.

Figure 3-7: Example system with RapidIO Interface



Refer to the RapidIO trade association web site's product list at rapidio.org for a list of processors that support a RapidIO interface.

Refer to the following documents for a complete description of the RapidIO MegaCore function: *RapidIO MegaCore Function User Guide* and *AN513: RapidIO Interoperability With TI 6482 DSP Reference Design*.

Related Information

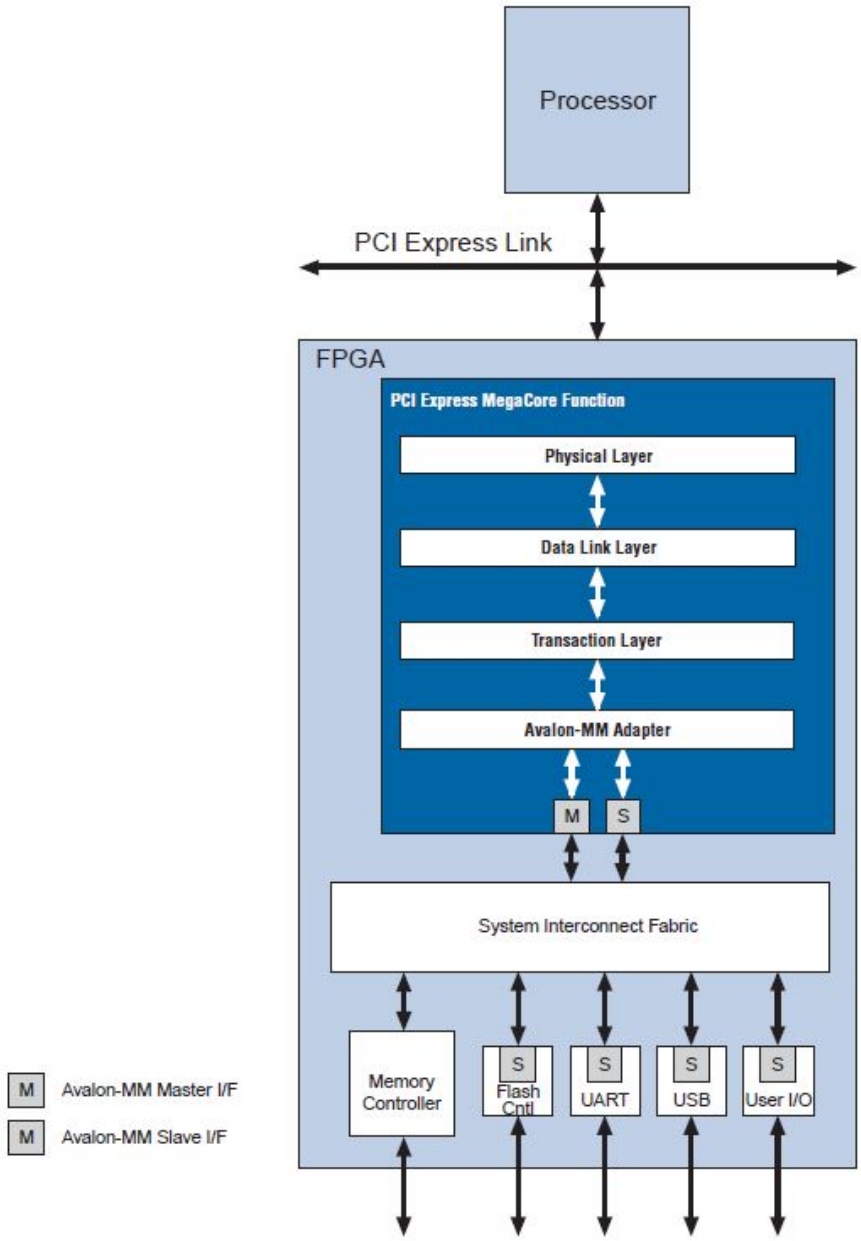
- **www.rapidio.org**
- **RapidIO MegaCore Function User Guide**
- **RapidIO Interoperability with TI 6482 DSP Reference Design**

PCI Express Interface

The Altera IP Compiler for PCI Express configured using the Qsys design flow uses the IP Compiler for PCI Express's Avalon-MM bridge module to connect the IP Compiler for PCI Express component to the system interconnect fabric. The bridge facilitates the design of PCI Express systems that use the Avalon-MM interface to access Qsys components. The figure below illustrates a design that links an external processor to an Qsys system using the IP Compiler for PCI Express.

You can also implement the IP Compiler for PCI Express using the Parameter Editor design flow. The configuration options for the two design flows are different. The IP Compiler for PCI Express is available in Stratix IV and Arria II GX devices as a hard IP implementation and can be used as a root port or end point. In Stratix V devices, Altera provides the Stratix V Hard IP for PCI Express.

Figure 3-8: Example System with PCI Express Interface



The figure shows an example system in which an external processor communicates with an Altera FPGA through a PCI Express link.

For more information about using the IP Compiler for PCI Express refer to the following reference documents:

Related Information

- **IP Compiler for PCI Express User Guide**
- **AN456: PCI Express High Performance Reference Design**

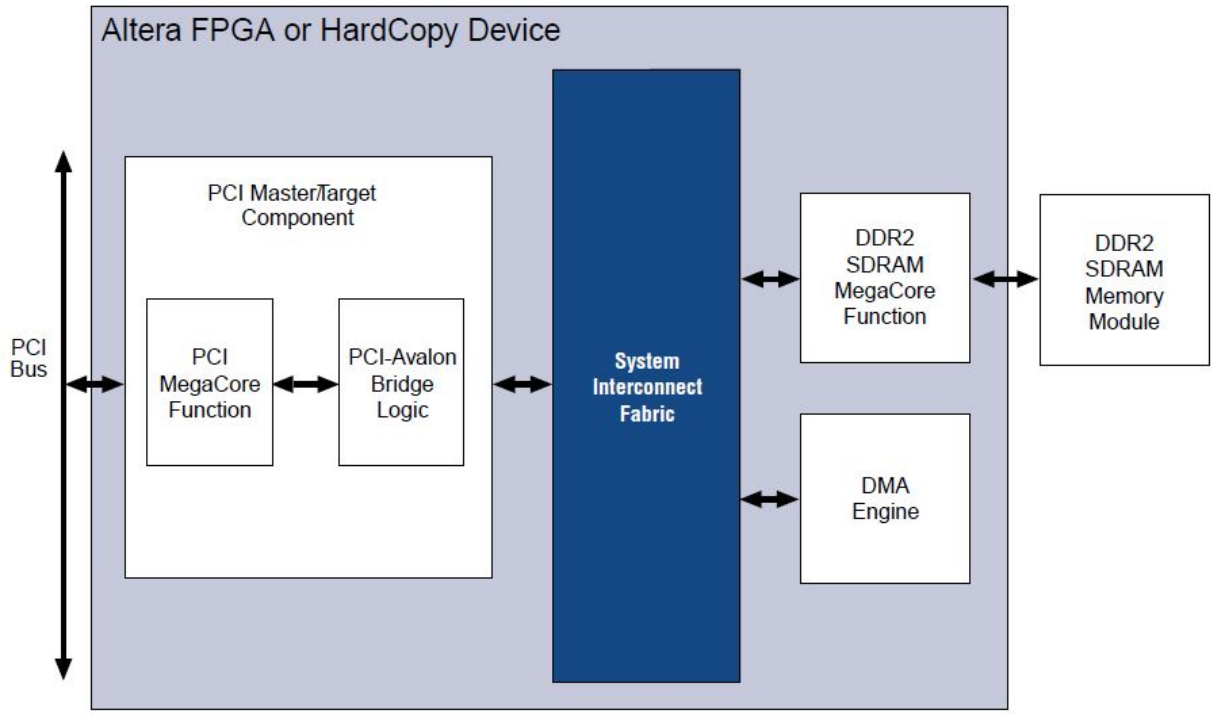
- External PHY Support in PCI Express MegaCore Functions
- PCI Express to External Memory Reference Design

PCI Interface

Altera offers a wide range of PCI local bus solutions that you can use to connect a host processor to an FPGA. You can implement the PCI MegaCore function using the Parameter Editor or Qsys design flow.

The PCI Qsys flow is an easy way to implement a complete Avalon-MM system which includes peripherals to expand system functionality without having to be well-acquainted with the Avalon-MM protocol. The figure below illustrates a Qsys system using the PCI MegaCore function. You can parameterize the PCI MegaCore function with a 32- or 64-bit interface.

Figure 3-9: PCI MegaCore Function in a Qsys System



For more information refer to the *PCI Compiler User Guide*.

Related Information

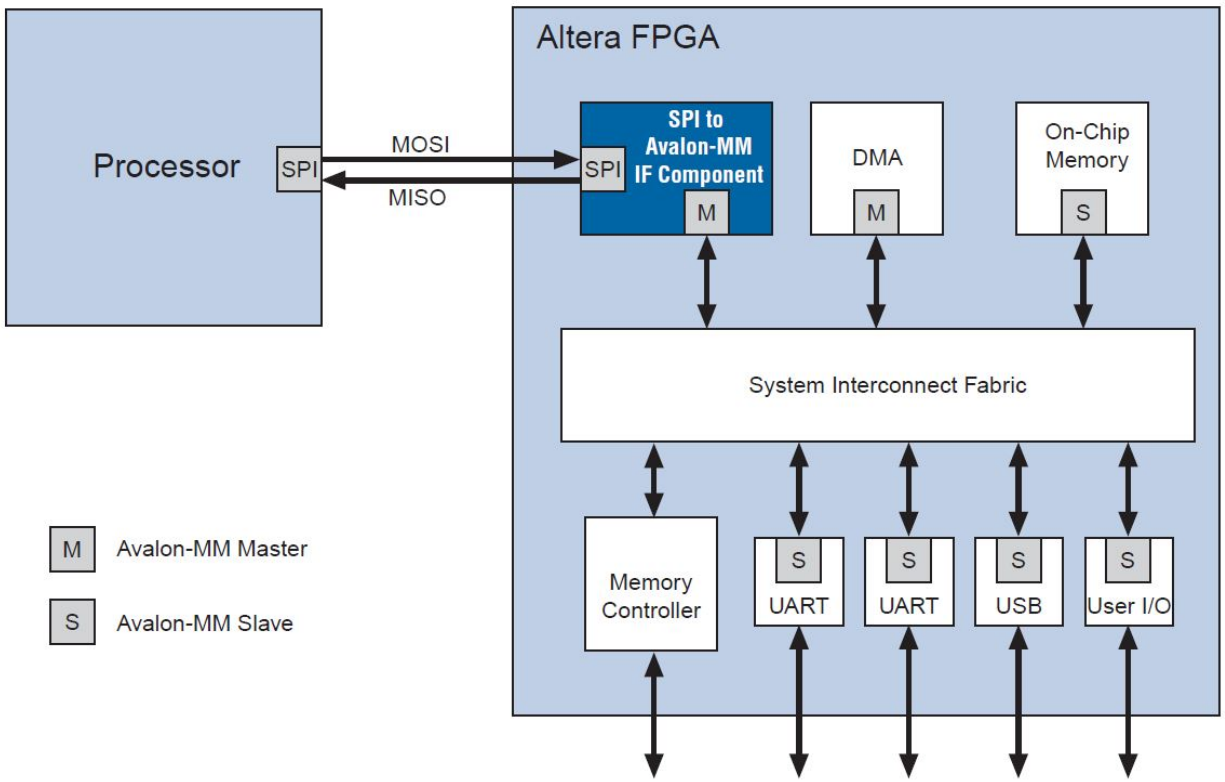
PCI Compiler User Guide

Serial Protocol Interface (SPI)

The SPI Slave to Avalon Master Bridge component provides a simple connection between processors and Qsys systems through a four-wire industry standard serial interface. Host systems can initiate Avalon-MM transactions by sending encoded streams of bytes through the core's serial interface. The core supports read and write transactions to the Qsys system for memory access and peripheral expansion.

The SPI Slave to Avalon Master Bridge is an Qsys-ready component that integrates easily into any Qsys system. Processors that include an SPI interface can easily encapsulate Avalon-MM transactions for reads and writes using the protocols outlined in the SPI Slave/JTAG to Avalon Master Bridge Cores chapter of the *Embedded Peripherals IP User Guide*.

Figure 3-10: Example System with SPI to Avalon-MM Interface Component



Details of each protocol layer can be found in the following chapters of the *Embedded Peripherals IP User Guide*:

SPI Slave/JTAG to Avalon Master Bridge Cores—Provide a connection from an external host system to an Qsys system. Allow an SPI master to initiate Avalon-MM transactions.

Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores—Provide a connection from an external host system to an Qsys system. Allow an SPI master to initiate Avalon-ST transactions.

Avalon Packets to Transactions Converter Core—Receives streaming data from upstream components and initiates Avalon-MM transactions. Returns Avalon-MM transaction responses to requesting components.

The SPI Slave to Avalon Master Bridge Design Example demonstrates SPI transactions between an Avalon-MM host system and a remote SPI system.

Related Information

- **Embedded Peripherals IP User Guide**
- **SPI Slave to Avalon Master Bridge Design Example**

Custom Bridge Interfaces

Many bus protocols can be mapped to the system interconnect fabric either directly or with some custom bridge interface logic to compensate for differences between the interface standards. The Avalon-MM interface standard, which Qsys supports, is a synchronous, memory-mapped interface that is easy to create custom bridges for.

If required, you can use the component editor available in Qsys to quickly define a custom bridge component to adapt the external processor bus to the Avalon-MM interface or any other standard interface that is defined in the *Avalon Interfaces Specifications*. The Templates menu available in the component editor includes menu items to add any of the standard Avalon interfaces to your custom bridge. You can then use the Interfaces tab of the component editor to modify timing parameters including: Setup, Read Wait, Write Wait, and Hold timing parameters, if required.

For more information about the component editor, refer to the Creating Qsys Components chapter of the *Quartus Prime Handbook Volume 1: Design and Synthesis*.

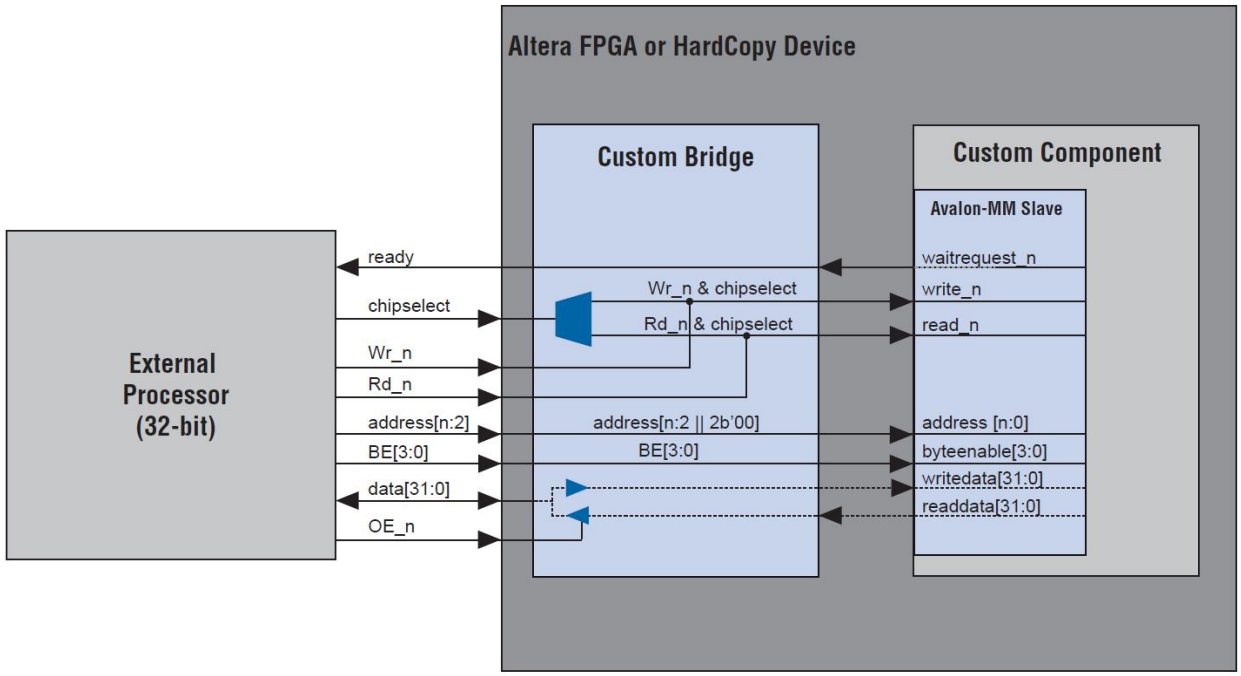
The Avalon-MM protocol requires that all masters provide byte addresses. Consequently, it may be necessary for your custom bridge component to add address wires when translating from the external processor bus interface to the Avalon-MM interface. For example, if your processor bus has a 16-bit word address, you must add one additional low-order address bit. If processor bus drives 32-bit word addresses, you must add two additional, low-order address bits. In both cases, the extra bits should be tied to 0. The external processor accesses individual byte lanes using the byte enable signals.

Consider the following points when designing a custom bridge to interface between an external processor and the Avalon-MM interface:

- The processor bus signals must comply or be adapted with logic to comply with the signals used for transactions, as described in the *Avalon Interfaces Specifications*.
- The external processor must support the Avalon waitrequest signal that inserts wait-state cycles for slave components
- The system bus must have a bus reference clock to drive Qsys interface logic in the FPGA.
- No time-out mechanism is available if you are using the Avalon-MM interface.
- You must analyze the timing requirements of the system. You should perform a timing analysis to guarantee that all synchronous timing requirements for the external processor and Avalon-MM interface are met. Examine the following timing characteristics:
  - Data t_{SU}, t_H , and t_{CO} times to the bus reference clock
  - f_{MAX} of the system matches the performance of the bus reference clock
  - Turn-around time for a read-to-write transfer or a write-to-read transfer for the processor is well understood

If your processor has dedicated read and write buses, you can map them to the Avalon-MM readdata and writedata signals. If your processor uses a bidirectional data bus, the bridge component can implement the tristate logic controlled by the processor's output enable signal to merge the readdata and writedata signals into a bidirectional data bus at the pins of the FPGA. Most of the other processor signals can pass through the bridge component if they adhere to the Avalon-MM protocol. The figure below illustrates the use of a bridge component with a 32-bit external processor.

Figure 3-11: Custom Bridge to Adapt an External Processor to an Avalon-MM Slave Interface



For more information about designing with the Avalon-MM interface refer to the *Avalon Interfaces Specifications*.

Related Information

- **Avalon Interface Specifications**
- **Creating Qsys Components**

Avalon-MM Byte Ordering

This section describes Avalon Memory-Mapped (Avalon-MM) interface bus byte ordering and provides recommendations for representing data in your system. Understanding byte ordering in both hardware and software is important when using intellectual property (IP) cores that interpret data differently.

Altera recommends understanding the following documents before proceeding:

- Qsys Interconnect chapter of the *Quartus Prime Handbook Volume 1: Design and Synthesis*
- The Avalon Interface Specifications

Related Information

- **Qsys Interconnect**
- **Avalon Interface Specifications**

Endianness

The term *endian* describes data byte ordering in both hardware and software. The two most common forms of data byte ordering are little endian and big endian. Little endian means that the least significant portion of a value is presented first and stored at the lowest address in memory. Big endian means the most significant portion of a value is presented first and stored at the lowest address in memory. For example, consider the value 0x1234. In little endian format, the 4 is the first digit presented or stored. In big endian format, the 1 is the first digit presented or stored.

Endianness typically refers only to byte ordering. Bit ordering within each byte is a separate subject covered in "PowerPC Bus Byte Ordering" and "ARM BE-32 Bus Byte Ordering".

Related Information

- **PowerPC Bus Byte Ordering** on page 3-27
- **ARM BE-32 Bus Byte Ordering** on page 3-29

Hardware Endianness

Hardware developers can map the data bits of an interface in any order. There must be coordination and agreement among developers so that the data bits of an interface correctly map to address offsets for any master or slave interface connected to the interconnect. Consistent hardware endianness or bus byte ordering is especially important in systems that contain IP interfaces of varying data widths because the interconnect performs the data width conversion between the master and slave interfaces. The key to simplifying bus byte ordering is to be consistent system-wide when connecting IP cores to the interconnect. For example, if all but one IP core in your system use little endian bus byte ordering, modify the interface of the one big endian IP core to conform to the rest of the system.

The way an IP core presents data to the interconnect is not dependent on the internal representation of the data within the core. IP cores can map the data interface to match the bus data byte ordering specification independent of the internal arithmetic byte ordering.

Software Endianness

Software endianness or *arithmetic byte ordering* is the internal representation of data within an IP core, software compiler, and peripheral drivers. Processors can treat byte offset 0 of a variable as the most or least significant byte of a multibyte word. For example, the value 0x0A0B0C0D, which spans multiple bytes, can be represented different ways. A little endian processor considers the byte 0x0D of the value to be located at the lowest byte offset in memory, whereas a big endian processor considers the byte 0x0A of the value to be located at the lowest byte offset in memory.

The example below shows a C code fragment that illustrates the difference between the little endian and big endian arithmetic byte ordering used by most processors.

Example 3-1: Reading Byte Offset 0 of a 32-Bit Word

```
long * long_ptr;
char byte_value;
*long_ptr = 0x0A0B0C0D; // 32-bit store to 'long_ptr'
byte_value = *((char *)long_ptr); // 8-bit read from 'long_ptr'
```

In the example, the processor writes the 32-bit value 0x0A0B0C0D to memory, then reads the first byte of the value back. A little endian processor such as the Nios II processor, which considers memory byte offset 0 to be the least significant byte of a word, stores byte 0x0D to byte offset 0 of pointer location `long_ptr`. A processor such as a PowerPC®, which considers memory byte offset 0 to be the most significant byte of a

word, stores byte 0x0A to byte offset 0 of pointer location `long_ptr`. As a result, the variable `byte_value` is loaded with 0x0D if this code executes on a little endian Nios II processor and 0x0A if this code executes on a big endian PowerPC processor.

Arithmetic byte ordering is not dependent on the bus byte ordering used by the processor data master that accesses memory. However, word and halfword accesses sometimes require byte swapping in software to correctly interpret the data internally by the processor.

For more information, refer to "Arithmetic Byte Reordering" and "System-Wide Arithmetic Byte Reordering in Software".

Related Information

- **Arithmetic Byte Reordering** on page 3-31
- **System-Wide Arithmetic Byte Reordering in Software** on page 3-34

Avalon-MM Interface Ordering

To ensure correct data communication, the Avalon-MM interface specification requires that each master or slave port of all components in your system pass data in descending bit order with data bits 7 down to 0 representing byte offset 0. This bus byte ordering is a little endian ordering. Any IP core that you add to your system must comply with the Avalon-MM interface specification. This ordering ensures that when any master accesses a particular byte of any slave port, the same physical byte lanes are accessed using a consistent bit ordering. For more information, refer to the Avalon Interface Specifications.

The interconnect handles dynamic bus sizing for narrow to wide and wide to narrow transfers when the master and slave port widths connected together do not match. When a wide master accesses a narrow slave, the interconnect serializes the data by presenting the lower bytes to the slave first. When a narrow master accesses a wide slave, the interconnect performs byte lane realignment to ensure that the master accesses the appropriate byte lanes of the slave.

For more information, refer to the Qsys Interconnect chapter of the *Quartus Prime Handbook Volume 1: Design and Synthesis*

Related Information

- **Qsys Interconnect**
- **Avalon Interface Specifications**

Dynamic Bus Sizing DMA Examples

A direct memory access (DMA) engine moves memory contents from a source location to a destination location. Because Qsys supports dynamic bus sizing, the data widths of the source and destination memory in the examples do not need to match the width of the DMA engine. The DMA engine reads data from a source base address and sequentially increases the address until the last read completes. The DMA engine also writes data to a destination base address and sequentially increases the address until the last write completes.

The following three figures illustrate example DMA transfers to and from memory of differing data widths. The source memory is populated with an increasing sequential pattern starting with the value 0 at base address 0. The DMA engine begins transferring data starting from the base address of the source memory to the base address of the destination memory. The interconnect always transfers the lower bytes first when any width adaptation takes place. The width adaptation occurs automatically within the interconnect.

Figure 3-12: 16-Bit to 32-Bit Memory DMA Transfer

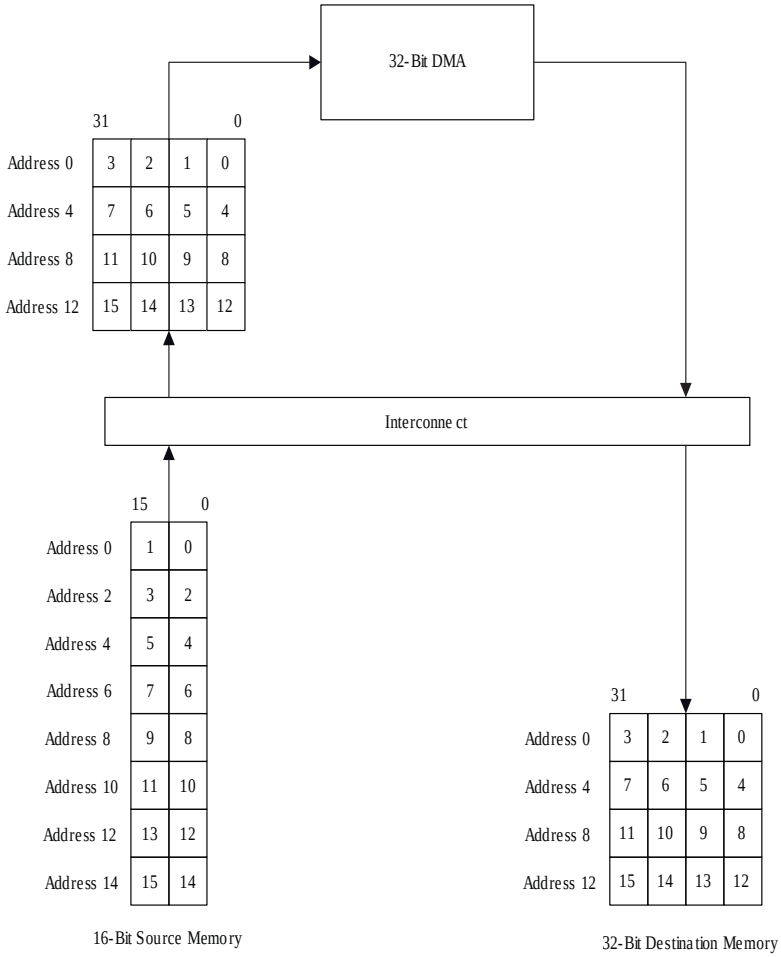


Figure 3-13: 32-Bit to 64-Bit Memory DMA Transfer

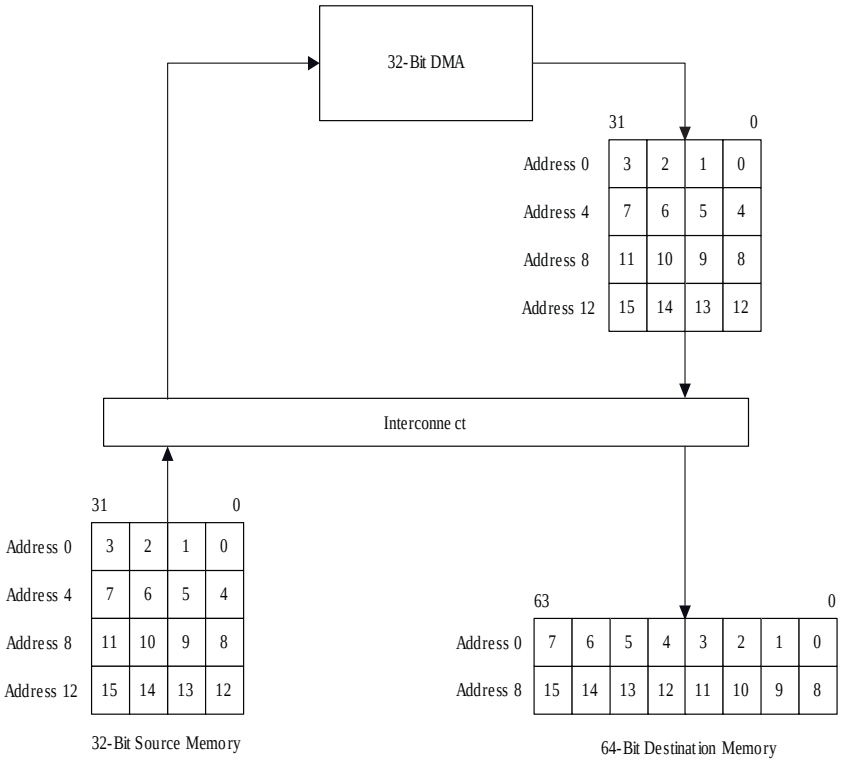
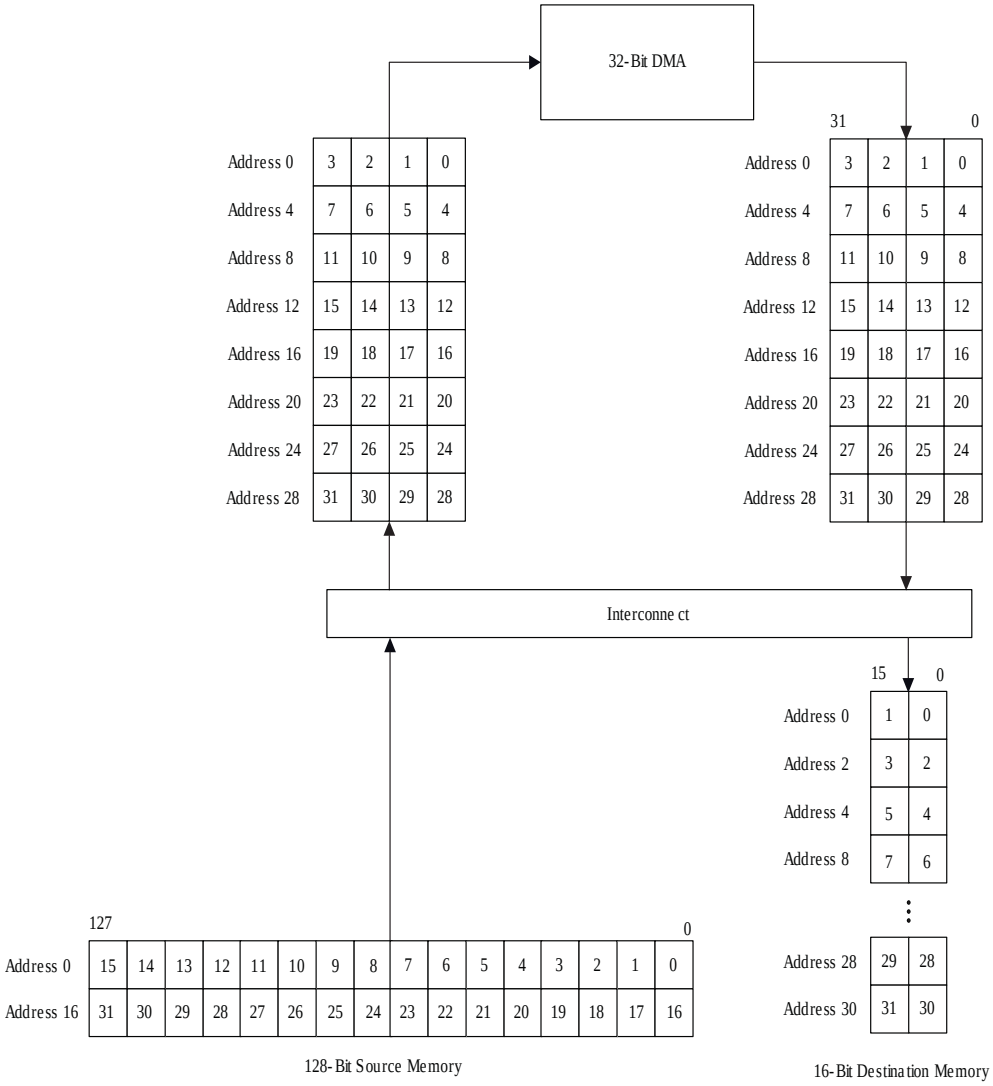


Figure 3-14: 128-Bit to 16-Bit Memory DMA Transfer

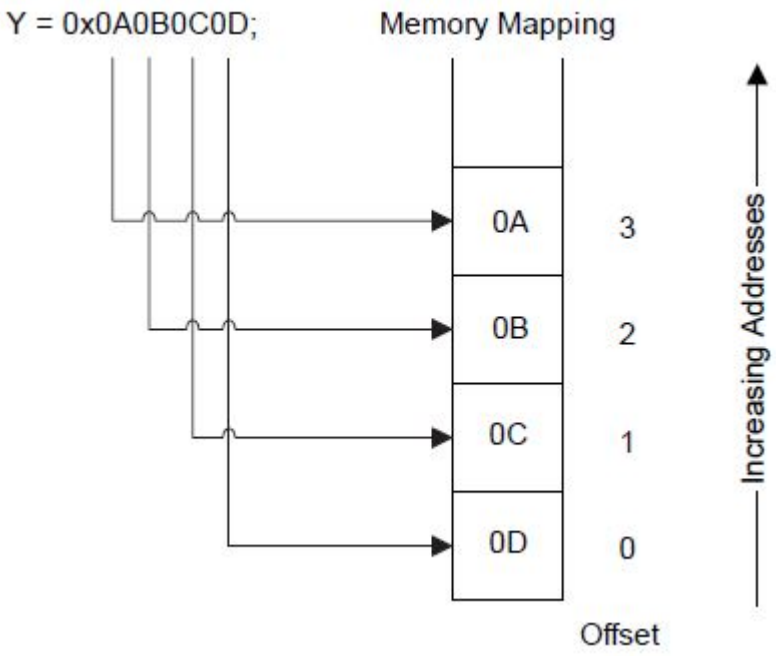


Nios II Processor Data Accesses

In the Nios II processor, the internal arithmetic byte ordering and the bus byte ordering are both little endian. Internally, the processor and its compiler map the least significant byte of a value to the lowest byte offset in memory.

For example, the figure below shows storing the 32-bit value 0x0A0B0C0D to the variable *Y*. The action maps the least significant byte 0x0D to offset 0 of the memory used to store the variable.

Figure 3-15: Nios II 32-Bit Byte Mapping



The Nios II processor is a 32-bit processor. For data larger than 32 bits, the same mapping of the least significant byte to lowest offset occurs. For example, if the value 0x0807060504030201 is stored to a 64-bit variable, the least significant byte 0x01 of the variable is stored to byte offset 0 of the variable in memory. The most significant byte 0x08 of the variable is stored to byte offset 7 of the variable in memory. The processor writes the least significant four bytes 0x04030201 of the variable to memory first, followed by the most significant four bytes 0x08070605.

The master interfaces of the Nios II processor comply with Avalon-MM bus byte ordering by providing read and write data to the interconnect in descending bit order with bits 7 down to 0 representing byte offset 0. Because the Nios II processor uses a 32-bit data path, the processor can access the interconnect with seven different aligned accesses. The table below shows the seven valid write accesses that the Nios II processor can present to the interconnect.

Table 3-3: Nios II Write Data Byte Mapping

Access Size (Bits)	Offset (Bytes)	Value	Byte Enable (Bits 3:0)	Write Data (Bits 31:24)	Write Data (Bits 23:16)	Write Data (Bits 15:8)	Write Data (Bits 7:0)
8	0	0x0A	0001	—	—	—	0x0A
8	1	0x0A	0010	—	—	0x0A	—
8	2	0x0A	0100	—	0x0A	—	—
8	3	0x0A	1000	0x0A	—	—	—
16	0	0x0A0B	0011	—	—	0x0A	0x0B
16	2	0x0A0B	1100	0x0A	0x0B	—	—

Access Size (Bits)	Offset (Bytes)	Value	Byte Enable (Bits 3:0)	Write Data (Bits 31:24)	Write Data (Bits 23:16)	Write Data (Bits 15:8)	Write Data (Bits 7:0)
32	0	0x0A0B0C0D	1111	0x0A	0x0B	0x0C	0x0D

The code fragment shown in the example generates all seven of the accesses described in the table in the order presented in the table, where BASE is a location in memory aligned to a four-byte boundary.

Example 3-2: Nios II Write Data Byte Mapping Code

```
IOWR_8DIRECT(BASE, 0, 0x0A);
IOWR_8DIRECT(BASE, 1, 0x0A);
IOWR_8DIRECT(BASE, 2, 0x0A);
IOWR_8DIRECT(BASE, 3, 0x0A);
IOWR_16DIRECT(BASE, 0, 0x0A0B);
IOWR_16DIRECT(BASE, 2, 0x0A0B);
IOWR_32DIRECT(BASE, 0, 0x0A0B0C0D);
```

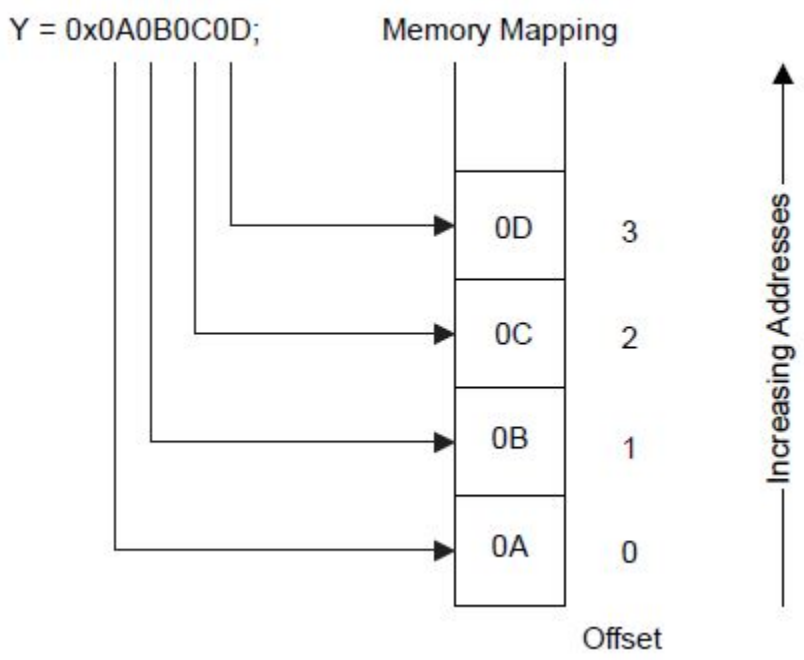
Adapting Processor Masters to be Avalon-MM Compliant

Because the way the Nios II processor presents data to the interconnect is Avalon-MM compliant, no extra effort is required to connect the processor to the interconnect. This section describes how to modify non-Avalon-MM compliant processor masters to achieve Avalon-MM compliance.

Some processors use a different arithmetic byte ordering than the Nios II processor uses, and as a result, typically use a different bus byte ordering than the Avalon-MM interface specification supports. When connecting one of these processors directly to the interconnect in a system containing other masters such as a Nios II processor, accesses to the same address result in accessing different physical byte lanes of the slave port. Mixing masters and slaves that conform to different bus byte ordering becomes nearly impossible to manage at a system level. These mixed bus byte ordering systems are difficult to maintain and debug. Altera requires that the master interfaces of any processors you add to your system are Avalon-MM compliant.

Processors that use a big endian arithmetic byte ordering, which is opposite to what the Nios II processor implements, map the most significant byte of the variable to the lowest byte offset of the variable in memory. For example, the figure below shows how a PowerPC processor core stores the 32-bit value 0x0A0B0C0D to the memory containing the variable Y. The PowerPC stores the most significant byte, 0x0A, to offset 0 of the memory containing the variable.

Figure 3-16: Power PC 32-Bit Byte Mapping



This arithmetic byte ordering is the opposite of the ordering shown in "Nios II Processor Data Accesses". Because the arithmetic byte ordering internal to the processor is independent of data bus byte ordering external to the processor, you can adapt processor masters with non-Avalon-MM compliant bus byte ordering to present Avalon-MM compliant data to the interconnect.

The following sections describe the bus byte ordering for the two most common processors that are not Avalon-MM complaint:

- "PowerPC Bus Byte Ordering"
- "ARM BE-32 Bus Byte Ordering"

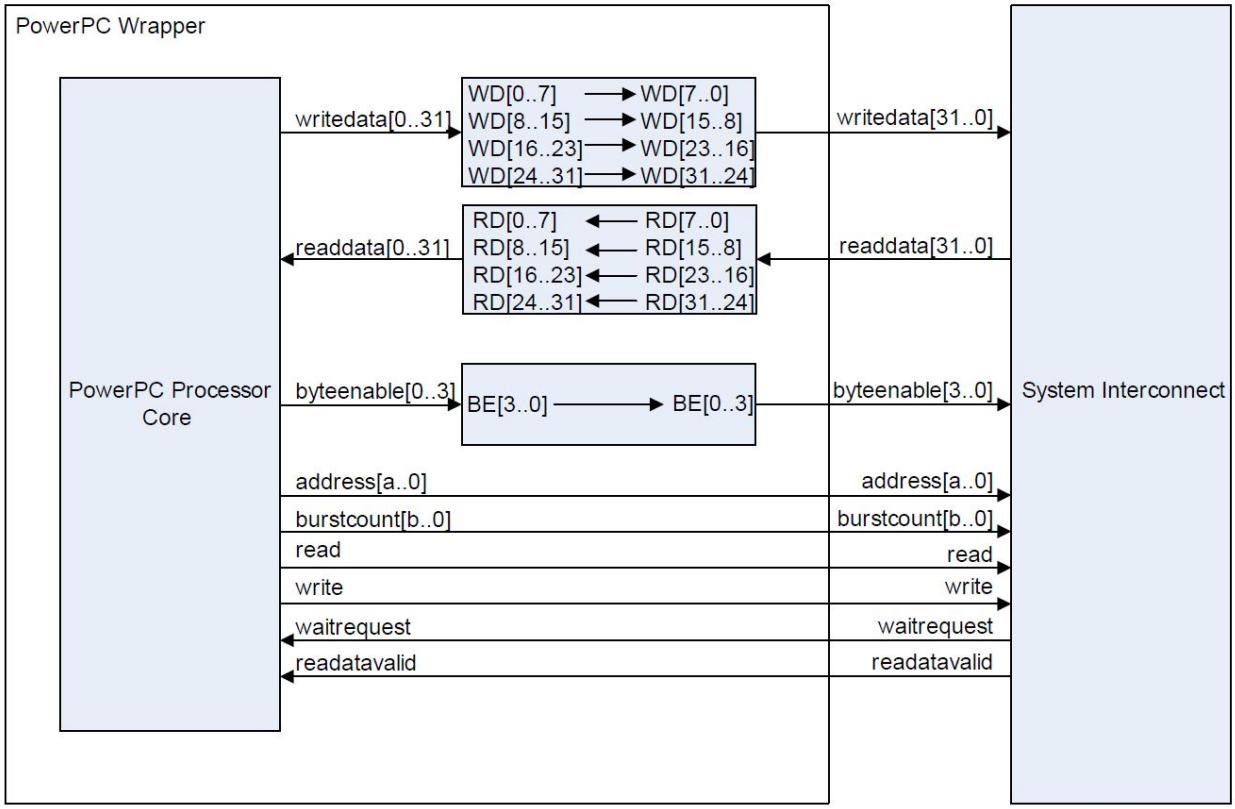
Related Information

- **Nios II Processor Data Accesses** on page 3-24
- **PowerPC Bus Byte Ordering** on page 3-27
- **ARM BE-32 Bus Byte Ordering** on page 3-29

PowerPC Bus Byte Ordering

The byte positions of the PowerPC bus byte ordering are aligned with the byte positions of the Avalon-MM interface specification; however, the bits within each byte are misaligned. PowerPC processor cores use an ascending bit ordering when the masters are connected to the interconnect. For example, a 32-bit PowerPC core labels the bus data bits 0 up to 31. A PowerPC core considers bits 0 up to 7 as byte offset 0. This layout differs from the Avalon-MM interface specification, which defines byte offset 0 as data bits 7 down to 0. To connect a PowerPC processor to the interconnect, you must rename the bits in each byte lane as shown below.

Figure 3-17: PowerPC Bit-Renaming Wrapper



In the figure above, bit 0 is renamed to bit 7, bit 1 is renamed to bit 6, bit 2 is renamed to bit 5, and so on. By renaming the bits in each byte lane, byte offset 0 remains in the lower eight data bits. You must rename the bits in each byte lane separately. Renaming the bits by reversing all 32 bits creates a result that is not Avalon-MM compliant. For example, byte offset 0 would shift to data bits 31 down to 24, not 7 down to 0 as required.

Note: Because the bits are simply renamed, this additional hardware does not occupy any additional FPGA resources nor impact the f_{MAX} of the data interface.

ARM BE-32 Bus Byte Ordering

Some ARM cores use a bus byte ordering commonly referred to as *big endian* 32 (BE-32). BE-32 processor cores use a descending bit ordering when the masters are connected to the interconnect. For example, an ARM BE-32 processor core labels the data bits 31 down to 0. Such a processor core considers bits 31 down to 24 as byte offset 0. This layout differs from the Avalon-MM specification, which defines byte 0 as data bits 7 down to 0.

A BE-32 processor core accesses memory using the bus mapping shown below.

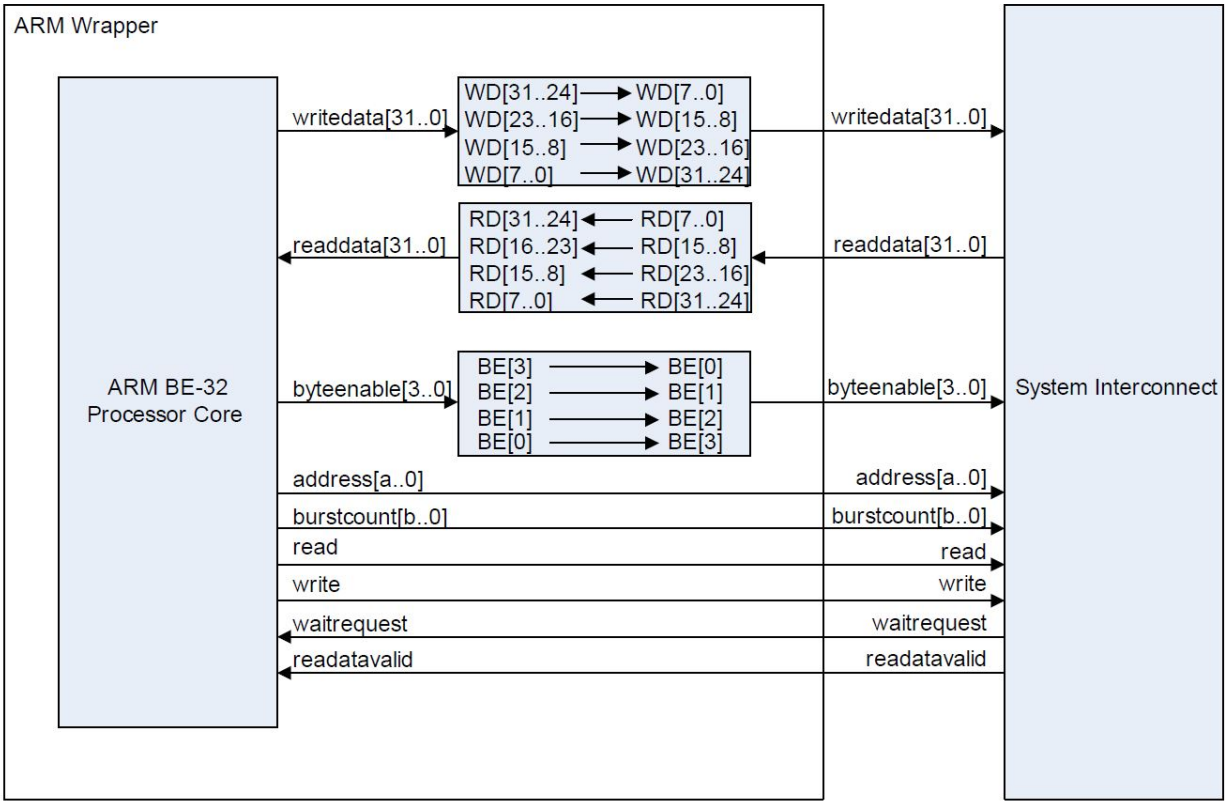
Table 3-4: ARM BE-32 Write Data Mapping

Access Size (Bits)	Offset (Bytes)	Value	Byte Enable (Bits 3:0)	Write Data (Bits 31:24)	Write Data (Bits 23:16)	Write Data (Bits 15:8)	Write Data (Bits 7:0)
8	0	0x0A	1000	0x0A	—	—	—
8	1	0x0A	0100	—	0x0A	—	—
8	2	0x0A	0010	—	—	0x0A	—
8	3	0x0A	0001	—	—	—	0x0A
16	0	0x0A0B	1100	0x0A	0x0B	—	—
16	2	0x0A0B	0011	—	—	0x0A	0x0B
32	0	0x0A0B0C0D	1111	0x0A	0x0B	0x0C	0x0D

The write access behavior of the BE-32 processor shown in the table above differs greatly from the Nios II processor behavior shown in **Table 3-3**. The only consistent access is the full 32-bit write access. In all the other cases, each processor accesses different byte lanes of the interconnect.

To connect a processor with BE-32 bus byte ordering to the interconnect, rename each byte lane as the figure below shows.

Figure 3-18: ARM BE-32 Byte-Renaming Wrapper



Note: As in the case of the PowerPC wrapper logic, the ARM BE-32 wrapper does not consume any FPGA logic resources or degrade the f_{MAX} of the interface.

ARM BE-8 Bus Byte Ordering

Newer ARM processor cores offer a mode called big endian 8 (BE-8). BE-8 processor master interfaces are Avalon-MM compliant. Internally, the BE-8 core uses a big endian arithmetic byte ordering; however, at the bus level, the core maps the data to the interconnect with the little endian orientation the Avalon-MM interface specification requires.

Note: This byte reordering sometimes requires special attention. For more information, refer to "Arithmetic Byte Reordering".

Related Information

Arithmetic Byte Reordering on page 3-31

Other Processor Bit and Byte Orders

There are numerous other ways to order the data leaving or entering a processor master interface. For those cases, the approach to achieving Avalon-MM compliance is the same. In general, apply the following three steps to any processor core to ensure Avalon-MM compliance:

1. Identify the bit order.
2. Identify the location of byte offset 0 of the master.
3. Create a wrapper around the processor core that renames the data signals so that byte 0 is located on data 7 down to 0, byte 1 is located on data 15 down to 8, and so on.

Arithmetic Byte Reordering

Altering your system to conform to Avalon-MM byte ordering modifies the internal arithmetic byte ordering of multibyte values as seen by the software. For example, an Avalon-MM compliant big endian processor core such as an ARM BE-8 processor accesses memory using the bus mapping shown below.

Table 3-5: ARM BE-8 Write Data Mapping

Access Size (Bits)	Offset (Bytes)	Value	Byte Enable (Bits 3:0)	Write Data (Bits 31:24)	Write Data (Bits 23:16)	Write Data (Bits 15:8)	Write Data (Bits 7:0)
8	0	0x0A	0001	—	—	—	0x0A
8	1	0x0A	0010	—	—	0x0A	—
8	2	0x0A	0100	—	0x0A	—	—
8	3	0x0A	1000	0x0A	—	—	—
16	0	0x0A0B	0011	—	—	0x0B	0x0A
16	2	0x0A0B	1100	0x0B	0x0A	—	—
32	0	0x0A0B0C0D	1111	0x0D	0x0C	0x0B	0x0A

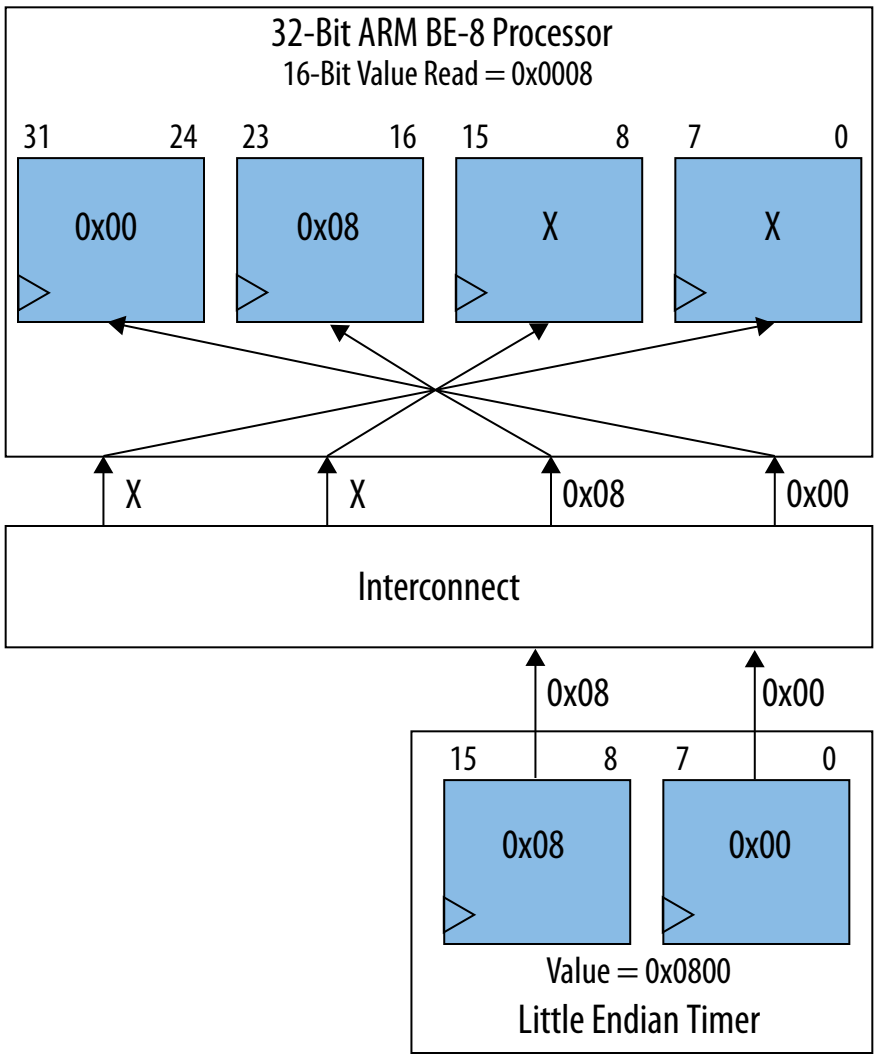
The big endian ARM BE-8 mapping in the table ablove matches the little endian Nios II processor mapping for all single byte accesses. If you ensure that your processor is Avalon-MM compliant, you can easily share individual bytes of data between big and little endian processors and peripherals.

However, making sure that the processor data master is Avalon-MM compliant only ensures that single byte accesses map to the same physical byte lanes of a slave port. In the case of multibyte accesses, the same byte lanes are accessed between the BE-8 and little endian processor; however, the value is not interpreted consistently. This mismatch is only important when the internal arithmetic byte ordering of the processor differs from other peripherals and processors in your system.

To correct the mismatch, you must perform arithmetic byte reordering in software for multibyte accesses. Interpretation of the data by the processor can vary based on the arithmetic byte ordering used by the processor and other processors and peripherals in the system.

For example, consider a 32-bit ARM BE-8 processor core that reads from a 16-bit little endian timer peripheral by performing a 16-bit read access. The ARM processor treats byte offset 0 as the most significant byte of any word. The timer treats byte offset 0 as the least significant byte of the 16-bit value. When the processor reads a value from the timer, the bytes of the value, as seen by software, are swapped. The figure below shows the swapping. A timer counter value of 0x0800 (2,048 clock ticks) is interpreted by the processor as 0x0008 (8 clock ticks) because the arithmetic byte ordering of the processor does not match the arithmetic byte ordering of the timer component.

Figure 3-19: ARM BE-8 Processor Accessing a Little Endian Peripheral



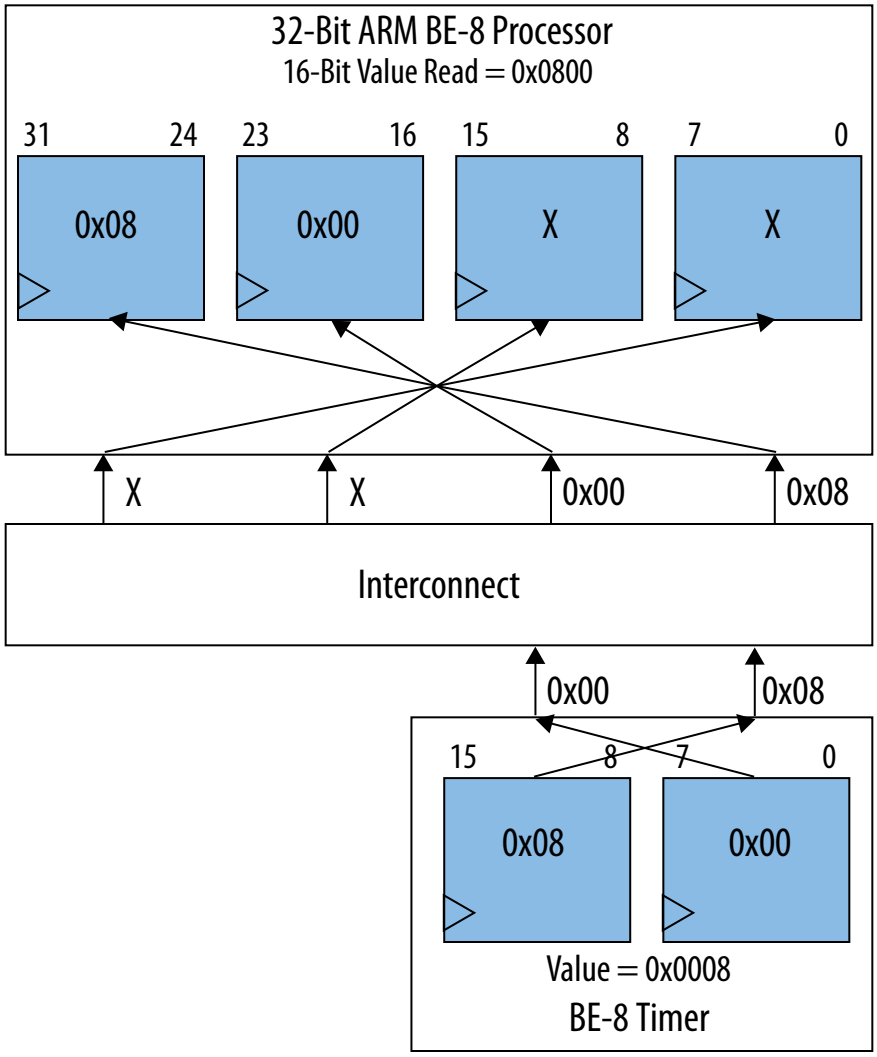
For the values to be interpreted accurately, the processor must either read each byte lane individually and then combine the two byte reads into a single 16-bit value in software, or read the single 16-bit value and swap the bytes in software.

The same issue occurs when you apply a bus-level renaming wrapper to an ARM BE-32 or PowerPC core. Both processor cores treat byte offset 0 as the most significant byte of any value. As a result, you must handle any mismatch between arithmetic byte ordering of data used by the processor and peripherals in your system.

On the other hand, if the timer in the figure above were to treat the most significant byte of the 16-bit value as byte 0 (big endian ordering), the data would arrive at the processor master in the same arithmetic byte ordering used by the processor. If the processor and the component internally implement the same arithmetic byte ordering, no software swapping of bytes is necessary for multibyte accesses.

The figure below shows how the value 0x0800 of a big endian timer is read by the processor. The value is retained without the need to perform any byte swapping in software after the read completes.

Figure 3-20: ARM BE-8 Processor Accessing a BE-8 Peripheral



System-Wide Design Recommendations

In the previous sections, we discussed arithmetic and bus byte ordering from a processor perspective. The same concepts directly apply to any component in your system. Any component containing Avalon-MM slave ports must also adhere to the Avalon-MM specification, which states that the data bits be defined in descending order with byte offset 0 positioned at bits 7 down to 0. As long as the component's slave port is Avalon-MM compliant, you can use any arithmetic byte ordering within the component.

System-Wide Arithmetic Byte Ordering

Typically, the most convenient arithmetic byte ordering to use throughout a system is the ordering the processor uses, if one is present. If the processor uses a different arithmetic byte ordering than the rest of the system, you must write software that rearranges the ordering for all multibyte accesses.

The majority of the IP provided by Altera that contains an Avalon-MM master or slave port uses little endian arithmetic byte ordering. If your system consists primarily of components provided by Altera, it is much easier to make the remainder of your system use the same little endian arithmetic byte ordering. When the entire system uses components that use the same arithmetic byte ordering and Avalon-MM bus byte ordering, arithmetic byte reordering within the processor or any component performing data accesses is not necessary.

Altera recommends writing your driver code to handle both big and little endian arithmetic byte ordering. For example, if the peripheral is little endian, write the peripheral driver to execute on both big and little endian processors. For little endian processors, no byte swapping is necessary. For big endian processors, all multibyte accesses requires a byte swap. Driver code selection is controlled at compile time or run time depending on the application and the peripheral.

System-Wide Arithmetic Byte Reordering in Software

If you cannot modify your system so that all the components use the same arithmetic byte ordering, you must implement byte reordering in software for multibyte accesses. Many processors today include instructions to accelerate this operation. If your processor does not have dedicated byte-reordering instructions, the example below shows how you can implement byte reordering in software by leveraging the macros for 16-bit and 32-bit data.

Example 3-3: Software Arithmetic Byte Reordering

```
/* Perform 16-bit byte reordering */
#define SW_16_BIT_ARITHMETIC_REORDERING (data) ( \
(((data) << 8) & 0xFF00) | \
(((data) >> 8) & 0x00FF) \
)

/* Perform 32-bit byte reordering */
#define SW_32_BIT_ARITHMETIC_REORDERING (data) ( \
(((data) << 24) & 0xFF000000) | \
(((data) << 8) & 0x00FF0000) | \
(((data) >> 8) & 0x0000FF00) | \
(((data) >> 24) & 0x000000FF) \
```

Choose the appropriate instruction or macro to perform the byte reordering based on the width of the value that requires arithmetic byte reordering. Because arithmetic byte ordering only applies to individual values stored in memory or peripherals, you must reverse the bytes of the value without disturbing the data stored in neighboring memory locations. For example, if you load a 16-bit value from a peripheral that uses a different arithmetic byte ordering, you must swap two bytes in software. If you attempt to load two 16-bit values as a packed 32-bit read access, you must swap the individual 16-bit values independently.

If you attempt to swap all four bytes at once, the two individual 16-bit values are swapped, which is not the original intent of the software developer.

Memory System Design

This section describes the efficient use of memories in a Qsys embedded systems. Efficient memory use increases the performance of FPGA-based embedded systems. Embedded systems use memories for a range of tasks, such as the storage of software code and lookup tables (LUTs) for hardware accelerators.

Memory Types

Your system's memory requirements depend heavily on the nature of the applications which you plan to run on the system. Memory performance and capacity requirements are small for simple, low cost systems. In contrast, memory throughput can be the most critical requirement in a complex, high performance system. The following general types of memories can be used in embedded systems.

Volatile Memory

A primary distinction in memory types is volatility. Volatile memories only hold their contents while power is applied to the memory device. As soon as power is removed, the memories lose their contents; consequently, volatile memories are unacceptable if data must be retained when the memory is switched off. Examples of volatile memories include static RAM (SRAM), synchronous static RAM (SSRAM), synchronous dynamic RAM (SDRAM), and FPGA on-chip memory.

Non-Volatile Memory

Non-volatile memories retain their contents when power is switched off, making them good choices for storing information that must be retrieved after a system power-cycle. Processor boot-code, persistent application settings, and FPGA configuration data are typically stored in non-volatile memory. Although non-volatile memory has the advantage of retaining its data when power is removed, it is typically much slower to write to than volatile memory, and often has more complex writing and erasing procedures. Non-volatile memory is also usually only guaranteed to be erasable a given number of times, after which it may fail. Examples of non-volatile memories include all types of flash, EPROM, and EEPROM. Most modern embedded systems use some type of flash memory for non-volatile storage.

Many embedded applications require both volatile and non-volatile memories because the two memory types serve unique and exclusive purposes. The following sections discuss the use of specific types of memory in embedded systems.

On-Chip Memory

On-chip memory is the simplest type of memory for use in an FPGA-based embedded system. The memory is implemented in the FPGA itself; consequently, no external connections are necessary on the circuit board. To implement on-chip memory in your design, simply select On-Chip Memory from the Component Library on the System Contents tab in Qsys. You can then specify the size, width, and type of on-chip memory, as well as special on-chip memory features such as dual-port access.

Advantages

On-chip memory is the highest throughput, lowest latency memory possible in an FPGA-based embedded system. It typically has a latency of only one clock cycle. Memory transactions can be pipelined, making a throughput of one transaction per clock cycle typical.

Some variations of on-chip memory can be accessed in dual-port mode, with separate ports for read and write transactions. Dual-port mode effectively doubles the potential bandwidth of the memory, allowing the memory to be written over one port, while simultaneously being read over the second port.

Another advantage of on-chip memory is that it requires no additional board space or circuit-board wiring because it is implemented on the FPGA directly. Using on-chip memory can often save development time and cost.

Finally, some variations of on-chip memory can be automatically initialized with custom content during FPGA configuration. This memory is useful for holding small bits of boot code or LUT data which needs to be present at reset.

Disadvantages

While on-chip memory is very fast, it is somewhat limited in capacity. The amount of on-chip memory available on an FPGA depends solely on the particular FPGA device being used, but capacities range from around 15 KBytes in the smallest Cyclone II device to just under 2 MBytes in the largest Stratix III device.

Because most on-chip memory is volatile, it loses its contents when power is disconnected. However, some types of on-chip memory can be initialized automatically when the FPGA is configured, essentially providing a kind of non-volatile function. For details, refer to the embedded memory chapter of the device handbook for the particular FPGA family you are using or Quartus® II Help.

Best Applications

The following sections describe the best uses of on-chip memory.

Cache

Because it is low latency, on-chip memory functions very well as cache memory for microprocessors. The Nios II processor uses on-chip memory for its instruction and data caches. The limited capacity of on-chip memory is usually not an issue for caches because they are typically relatively small.

Tightly Coupled Memory

The low latency access of on-chip memory also makes it suitable for tightly coupled memories. Tightly coupled memories are memories which are mapped in the normal address space, but have a dedicated interface to the microprocessor, and possess the high speed, low latency properties of cache memory.

For more information regarding tightly-coupled memories, refer to the *Using Tightly Coupled Memory with the Nios II Processor Tutorial*.

Related Information

Using Tightly Coupled Memory with the Nios II Processor Tutorial

Look Up Tables

For some software programming functions, particularly mathematical functions, it is sometimes fastest to use a LUT to store all the possible outcomes of a function, rather than computing the function in software. On-chip memories work well for this purpose as long as the number of possible outcomes fits reasonably in the capacity of on-chip memory available.

FIFO

Embedded systems often need to regulate the flow of data from one system block to another. FIFOs can buffer data between processing blocks that run most efficiently at different speeds. Depending on the size

of the FIFO your application requires, on-chip memory can serve as very fast and convenient FIFO storage.

For more information regarding FIFO buffers, refer to the On-Chip FIFO Memory Core chapter of the *Embedded Peripheral IP User Guide*.

Related Information

On-Chip FIFO Memory Core

Poor Applications

On-chip memory is poorly suited for applications which require large memory capacity. Because on-chip memory is relatively limited in capacity, avoid using it to store large amounts of data; however, some tasks can take better advantage of on-chip memory than others. If your application utilizes multiple small blocks of data, and not all of them fit in on-chip memory, you should carefully consider which blocks to implement in on-chip memory. If high system performance is your goal, place the data which is accessed most often in on-chip memory cache.

On-Chip Memory Types

Depending on the type of FPGA you are using, several types of on-chip memory are available. For details on the different types of on-chip memory available to you, refer to the device handbook for the particular FPGA family you are using.

Best Practices

To optimize the use of the on-chip memory in your system, follow these guidelines:

- Set the on-chip memory data width to match the data-width of its primary system master. For example, if you are connecting the on-chip memory to the data master of a Nios II processor, you should set the data width of the on-chip memory to 32 bits, the same as the data-width of the Nios II data master. Otherwise, the access latency could be longer than one cycle because the system interconnect fabric performs width translation.
- If more than one master connects to an on-chip memory component, consider enabling the dual-port feature of the on-chip memory. The dual-port feature removes the need for arbitration logic when two masters access the same on-chip memory. In addition, dual-ported memory allows concurrent access from both ports, which can dramatically increase efficiency and performance when the memory is accessed by two or more masters. However, writing to both slave ports of the RAM can result in data corruption if there is not careful coordination between the masters.

To minimize FPGA logic and memory utilization, follow these guidelines:

- Choose the best type of on-chip memory for your application. Some types are larger capacity; others support wider data-widths. The embedded memory section in the device handbook for the appropriate FPGA family provides details on the features of on-chip memories.
- Choose on-chip memory sizes that are a power of 2 bytes. Implementing memories with sizes that are not powers of 2 can result in inefficient memory and logic use.

External SRAM

The term external SRAM refers to any static RAM (SRAM) device that you connect externally to a FPGA. There are several varieties of external SRAM devices. The choice of external SRAM and its type depends on the nature of the application. Designing with SRAM memories presents both advantages and disadvantages.

Advantages

External SRAM devices provide larger storage capacities than on-chip memories, and are still quite fast, although not as fast as on-chip memories. Typical external SRAM devices have capacities ranging from around 128 KBytes to 10 MBytes. Specialty SRAM devices can even be found in smaller and larger capacities. SRAMs are typically very low latency and high throughput devices, slower than on-chip memory only because they connect to the FPGA over a shared, bidirectional bus. The SRAM interface is very simple, making connecting to an SRAM from an FPGA a simple design task. You can also share external SRAM buses with other external SRAM devices, or even with external memories of other types, such as flash or SDRAM.

Disadvantages

The primary disadvantages of external SRAM in an FPGA-based embedded system are cost and board real estate. SRAM devices are more expensive per MByte than other high-capacity memory types such as SDRAM. They also consume more board space per MByte than both SDRAM and FPGA on-chip memory, which consumes none.

Best Applications

External SRAM is quite effective as a fast buffer for medium-size blocks of data. You can use external SRAM to buffer data that does not fit in on-chip memory and requires lower latency than SDRAM provides. You can also group multiple SRAM memories to increase capacity.

SRAM is also optimal for accessing random data. Many SRAM devices can access data at non-sequential addresses with the same low latency as sequential addresses, an area where SDRAM performance suffers. SRAM is the ideal memory type for a large LUT holding the data for a color conversion algorithm that is too large to fit in on-chip memory, for example.

External SRAM performs relatively well when used as execution memory for a processor with no cache. The low latency properties of external SRAM help improve processor performance if the processor has no cache to mask the higher latency of other types of memory.

Poor Applications

Poor uses for external SRAM include systems which require large amounts of storage and systems which are cost-sensitive. If your system requires a block of memory larger than 10 MBytes, you may want to consider a different type of memory, such as SDRAM, which is less expensive.

External SRAM Types

There are several types of SRAM devices. The following types are the most popular:

- Asynchronous SRAM—This is the slowest type of SRAM because it is not dependent on a clock.
- Synchronous SRAM (SSRAM)—Synchronous SRAM operates synchronously to a clock. It is faster than asynchronous SRAM but also more expensive.
- Pseudo-SRAM—Pseudo-SRAM (PSRAM) is a type of dynamic RAM (DRAM) which has an SSRAM interface.
- ZBT SRAM—ZBT (zero bus turnaround) SRAM can switch from read to write transactions with zero turnaround cycles, making it very low latency. ZBT SRAM typically requires a special controller to take advantage of its low latency features.

Best Practices

To get the best performance from your external SRAM devices, follow these guidelines:

- Use SRAM interfaces which are the same data width as the data width of the primary system master that accesses the memory.
- If pin utilization or board real estate is a larger concern than the performance of your system, you can use SRAM devices with a smaller data width than the masters that will access them to reduce the pin count of your FPGA and possibly the number of memory devices on the PCB. However, this change results in reduced performance of the SRAM interface.

Flash Memory

Flash memory is a non-volatile memory type used frequently in embedded systems. In FPGA-based embedded systems, flash memory is always external because FPGAs do not contain flash memory. Because flash memory retains its contents after power is removed, it is commonly used to hold microprocessor boot code as well as any data which needs to be preserved in the case of a power failure. Flash memories are available with either a parallel or a serial interface. The fundamental storage technology for parallel and serial flash devices is the same.

Unlike SRAM, flash memory cannot be updated with a simple write transaction. Every write to a flash device uses a write command consisting of a fixed sequence of consecutive read and write transactions. Before flash memory can be written, it must be erased. All flash devices are divided into some number of erase blocks, or sectors, which vary in size, depending on the flash vendor and device size. Entire sections of flash must be erased as a unit; individual words cannot be erased. These requirements sometimes make flash devices difficult to use.

Advantages

The primary advantage of flash memory is that is non-volatile. Modern embedded systems use flash memory extensively to store not only boot code and settings, but large blocks of data such as audio or video streams. Many embedded systems use flash memory as a low power, high reliability substitute for a hard drive.

Among other non-volatile types of memory, flash memory is the most popular for the following four reasons:

- It is durable.
- It is erasable.
- It permits a large number of erase cycles.
- It is low-cost.

You can share flash buses with other flash devices, or even with external memories of other types, such as external SRAM or SDRAM.

Disadvantages

A major disadvantage of flash is its write speed. Because you can only write to flash devices using special commands, multiple bus transactions are required for each flash write. Furthermore, the actual write time, after the write command is sent, can be several microseconds. Depending on clock speed, the actual write

time can be in the hundreds of clock cycles. Because of the sector-erase restriction, if you need to change a data word in the flash, you must complete the following steps:

1. Copy the entire contents of the sector into a temporary buffer.
2. Erase the sector.
3. Change the single data word in the temporary buffer.
4. Write the temporary buffer back to the flash memory device.

This procedure contributes to the poor write speed of flash memory devices. Because of its poor write speed, flash memory is typically used only for storing data which must be preserved after power is turned off.

Typical Applications

Flash memory is effective for storing any data that you wish to preserve if power is removed from the system. Common uses of flash memory include storage of the following types of data:

- Microprocessor boot code
- Microprocessor application code to be copied to RAM at system startup
- Persistent system settings, including the following types of settings:
  - Network MAC address
  - Calibration data
  - User preferences
- FPGA configuration images
- Media (audio, video)

Poor Applications

Because of flash memory's slow write speeds, you should not use it for anything that does not need to be preserved after power-off. SRAM is a much better alternative if volatile memory is an option. Systems that use flash memory usually also include some SRAM as well.

One particularly poor use of flash is direct execution of microprocessor application code. If any of the code's writeable sections are located in flash memory, the software simply will not work, because flash memory cannot be written without using its special write commands. Systems that store application code in flash memory usually copy the application to SRAM before executing it.

Flash Types

There are several types of flash devices. The following types are the most popular:

- Serial flash – This flash has a serial interface to preserve device pins and board space. Because many serial flash devices have their own specific interface protocol, it is best to thoroughly read a serial flash device's datasheet before choosing it. Altera EPCS configuration devices are a type of serial flash.

  For more information about EPCS configuration devices, refer to the Documentation: Configuration Devices page on the Altera website.
- NAND flash – NAND flash can achieve very high capacities, up to multiple GBytes per device. The interface to NAND flash is a bit more complicated than that of CFI flash. It requires either a special controller or intelligent low-level driver software. You can use NAND Flash with Altera FPGAs; however, Altera does not provide any built-in support.

Related Information

- **Documentation: Configuration Devices**

- **Nios II Flash Programmer User Guide**

SDRAM

SDRAM is another type of volatile memory. It is similar to SRAM, except that it is dynamic and must be refreshed periodically to maintain its content. The dynamic memory cells in SDRAM are much smaller than the static memory cells used in SRAM. This difference in size translates into very high-capacity and low-cost memory devices.

In addition to the refresh requirement, SDRAM has other very specific interface requirements which typically necessitate the use of special controller hardware. Unlike SRAM, which has a static set of address lines, SDRAM divides up its memory space into banks, rows, and columns. Switching between banks and rows incurs some overhead, so that efficient use of SDRAM involves the careful ordering of accesses. SDRAM also multiplexes the row and column addresses over the same address lines, which reduces the pin count necessary to implement a given size of SDRAM. Higher speed varieties of SDRAM such as DDR, DDR2, and DDR3 also have strict signal integrity requirements which need to be carefully considered during the design of the PCB.

SDRAM devices are among the least expensive and largest-capacity types of RAM devices available, making them one of the most popular. Most modern embedded systems use SDRAM. A major part of an SDRAM interface is the SDRAM controller. The SDRAM controller manages all the address-multiplexing, refresh and row and bank switching tasks, allowing the rest of the system to access SDRAM without knowledge of its internal architecture.

For information about the SDRAM controllers available for use in Altera FPGAs, refer to the *External Memory Interface Handbook*.

Related Information

External Memory Interface Handbook Volume 2: Design Guidelines

Advantages

SDRAM's most significant advantages are its capacity and cost. No other type of RAM combines the low cost and large capacity of SDRAM, which makes it a very popular choice. SDRAM also makes efficient use of pins. Because row and column addresses are multiplexed over the same address pins, fewer pins are required to implement a given capacity of memory. Finally, SDRAM generally consumes less power than an equivalent SRAM device.

In some cases, you can also share SDRAM buses between multiple SDRAM devices, or even with external memories of other types, such as external SRAM or flash memory.

Disadvantages

Along with the high capacity and low cost of SDRAM, come additional complexity and latency. The complexity of the SDRAM interface requires that you always use an SDRAM controller to manage SDRAM refresh cycles, address multiplexing, and interface timing. Such a controller consumes FPGA logic elements that would normally be available for other logic.

SDRAM suffers from a significant amount of access latency. Most SDRAM controllers take measures to minimize the amount of latency, but SDRAM latency is always greater than that of regular external SRAM or FPGA on-chip memory. However, while first-access latency is high, SDRAM throughput can actually be quite high after the initial access latency is overcome, because consecutive accesses can be pipelined. Some types of SDRAM can achieve higher clock frequencies than SRAM, further improving throughput. The SDRAM interface specification also employs a burst feature to help improve overall throughput.

Best Applications

SDRAM is generally a good choice in the following circumstances:

- Storing large blocks of data—SDRAM's large capacity makes it the best choice for buffering large blocks of data such as network packets, video frame buffers, and audio data.
- Executing microprocessor code—SDRAM is commonly used to store instructions and data for microprocessor software, particularly when the program being executed is large. Instruction and data caches improve performance for large programs. Depending on the system topography and the SDRAM controller used, the sequential read patterns typical of cache line fills can potentially take advantage of SDRAM's pipeline and burst capabilities.

Poor Applications

SDRAM may not be the best choice in the following situations:

- Whenever low-latency memory access is required—Although high throughput is possible using SDRAM, its first-access latency is quite high. If low latency access to a particular block of data is a requirement of your application, SDRAM is probably not a good candidate to store that block of data.
- Small blocks of data—When only a small amount of storage is needed, SDRAM may be unnecessary. An on-chip memory may be able to meet your memory requirements without adding another memory device to the PCB.
- Small, simple embedded systems—If your system uses a small FPGA in which logic resources are scarce and your application does not require the capacity that SDRAM provides, you may prefer to use a small external SRAM or on-chip memory rather than devoting FPGA logic elements to an SDRAM controller.

SDRAM Types

There are a several types of SDRAM devices. The following types are the most common:

- SDR SDRAM—Single data rate (SDR) SDRAM is the original type of SDRAM. It is referred to as SDRAM or as SDR SDRAM to distinguish it from newer, double data rate (DDR) types. The name single data rate refers to the fact that a maximum of one word of data can be transferred per clock cycle. SDR SDRAM is still in wide use, although newer types of DDR SDRAM are becoming more common.
- DDR SDRAM—Double data rate (DDR) SDRAM is a newer type of SDRAM that supports higher data throughput by transferring a data word on both the rising and falling edge of the clock. DDR SDRAM uses 2.5 V SSTL signaling. The use of DDR SDRAM requires a custom memory controller.
- DDR2 SDRAM—DDR2 SDRAM is a newer variation of standard DDR SDRAM memory which builds on the success of DDR by implementing slightly improved interface requirements such as lower power 1.8 V SSTL signaling and on-chip signal termination.
- DDR3 SDRAM—DDR3 is another variant of DDR SDRAM which improves the potential bandwidth of the memory further by improving signal integrity and increasing clock frequencies.
- QDR, QDR II, and QDR II+ SRAM—Quad Data Rate (QDR) SRAM has independent read and write ports that run concurrently at double data rate. QDR SRAM is true dual-port (although the address bus is still shared), which gives this memory a high bandwidth, allowing back-to-back transactions without the contention issues that can occur when using a single bidirectional data bus. Write and read operations share address ports.

- RLDRAM II and RLDRAM 3—Reduced latency DRAM (RLDRAM) provides DRAM-based point-to-point memory devices designed for communications, imaging, server systems, networking, and cache applications requiring high density, high memory bandwidth, and low latency. The fast random access speeds in RLDRAM devices make them a viable alternative to SRAM devices at a lower cost.
- LPDDR2—LPDDR2-S is a high-speed SDRAM device internally configured as a 4- or 8-bank memory. All LPDDR2 devices use double data rate architecture on the address and command bus to reduce the number of input pins in the system. The 10-bit address and command bus contains command, address, and bank/row buffer information. Each command uses one clock cycle, during which command information is transferred on both the positive and negative edges of the clock.
- LPDDR3—LPDDR3-SDRAM is a high-speed synchronous DRAM device internally configured as an 8-bank memory. All LPDDR3 devices use double data rate architecture on the address and command bus to reduce the number of input pins in the system. The 10-bit address and command bus contains command, address, and bank buffer information. Each command uses one clock cycle, during which command information is transferred on both the positive and negative edges of the clock.

For more information about SDRAM types refer to the *External Memory Interface Handbook Volume 2: Design Guidelines*.

SDRAM Controller Types Available From Altera

The table below lists the SDRAM controllers that Altera provides. These SDRAM controllers are available without licenses.

Table 3-6: Memory Controller Available from Altera

Controller Name	Description
SDR SDRAM Controller	This controller is the only SDR SDRAM controller Altera offers. It is a simple, easy-to-use controller that works with most available SDR SDRAM devices.
DDR/DDR2 Controller Megacore Function	This controller is a legacy component which is maintained for existing designs only. Altera does not recommend it for new designs.
High Performance DDR/DDR2 Controller	This controller is the DDR/DDR2 controller that Altera recommends for new designs. It supports two primary clocking modes, full-rate and half-rate. • Full-rate mode presents data to the Qsys system at twice the width of the actual DDR SDRAM device at the full SDRAM clock rate. • Half-rate mode presents data to the Qsys system at four times the native SDRAM device data width at half the SDRAM clock rate.
High Performance DDR3 Controller	This controller is the DDR3 controller that Altera recommends for new designs. It is similar to the high performance DDR/DDR2 controller. It also supports full- and half-rate clocking modes.
Hard Memory Controller (HMC)	The hard memory controller initializes, refreshes, manages, and communicates with the external memory device. The HMC supports all the popular and emerging memory standards including DDR4, DDR3, and LPDDR3.

For more information about the available SDRAM controllers, refer to the *External Memory Interface Handbook Volume 3: Reference Material*.

Related Information

External Memory Interface Handbook Volume 3: Reference Material

Best Practices

When using the high performance DDR or DDR2 SDRAM controller, it is important to determine whether full-rate or half-rate clock mode is optimal for your application.

Half-Rate Mode

Half-rate mode is optimal in cases where you require the highest possible SDRAM clock frequency, or when the complexity of your system logic means that you are not able to achieve the clock frequency you need for the DDR SDRAM. In half-rate mode, the internal Avalon interface to the SDRAM controller runs at half the external SDRAM frequency.

In half-rate mode, the local data width (the data width inside the Qsys system) of the SDRAM controller is four times the data width of the physical DDR SDRAM device. For example, if your SDRAM device is 8 bits wide, the internal Avalon data port of the SDRAM controller is 32 bits. This design choice facilitates bursts of four accesses to the SDRAM device.

Full-Rate Mode

In full-rate mode, the internal Avalon interface to the SDRAM controller runs at the full external DDR SDRAM clock frequency. Use full-rate mode if your system logic is simple enough that it can easily achieve DDR SDRAM clock frequencies, or when running the system logic at half the clock rate of the SDRAM interface is too slow for your requirements.

When using full-rate mode, the local data width of the SDRAM controller is twice the data width of the physical DDR SDRAM. For example, if your SDRAM device is 16 bits wide, the internal Avalon data port of the SDRAM controller in full-rate mode is 32 bits wide. Again, this choice facilitate bursts to the SDRAM device.

Sequential Access

SDRAM performance benefits from sequential accesses. When access is sequential, data is written or read from consecutive addresses and it may be possible to increase throughput by using bursting. In addition, the SDRAM controller can optimize the accesses to reduce row and bank switching. Each row or bank change incurs a delay, so that reducing switching increases throughput.

Bursting

SDRAM devices employ bursting to improve throughput. Bursts group a number of transactions to sequential addresses, allowing data to be transferred back-to-back without incurring the overhead of requests for individual transactions. If you are using the high performance DDR/DDR2 SDRAM controller, you may be able to take advantage of bursting in the system interconnect fabric as well. Bursting is only useful if both the master and slave involved in the transaction are burst-enabled. Refer to the documentation for the master in question to check whether bursting is supported.

Selecting the burst size for the high performance DDR/DDR2 SDRAM controller depends on the mode in which you use the controller. In half-rate mode, the Avalon-MM data port is four times the width of the actual SDRAM device; consequently, four transactions are initiated to the SDRAM device for each single transfer in the system interconnect fabric. A burst size of four is used for those four transactions to SDRAM. This is the maximum size burst supported by the high performance DDR/DDR2 SDRAM controller. Consequently, using bursts for the high performance DDR/DDR2 SDRAM controller in half-rate mode does not increase performance because the system interconnect fabric is already using its maximum supported burst-size to carry out each single transaction.

However, in full-rate mode, you can use a burst size of two with the high performance DDR/DDR2 SDRAM controller. In full-rate mode, each Avalon transaction results in two SDRAM device transactions, so two Avalon transactions can be combined in a burst before the maximum supported SDRAM controller burst size of four is reached.

SDRAM Minimum Frequency

Many SDRAM devices, particularly DDR, DDR2, and DDR3 devices have minimum clock frequency requirements. The minimum clock rate depends on the particular SDRAM device. Refer to the datasheet of the SDRAM device you are using to find the device's minimum clock frequency.

SDRAM Device Speed

SDRAM devices, both SDR and DDR, come in several speed grades. When using SDRAM with FPGAs, the operating frequency of the FPGA system is usually lower than the maximum capability of the SDRAM

device. Therefore, it is typically not worth the extra cost to use fast speed-grade SDRAM devices. Before committing to a specific SDRAM device, consider both the expected SDRAM frequency of your system, and the maximum and minimum operating frequency of the particular SDRAM device.

Case Study

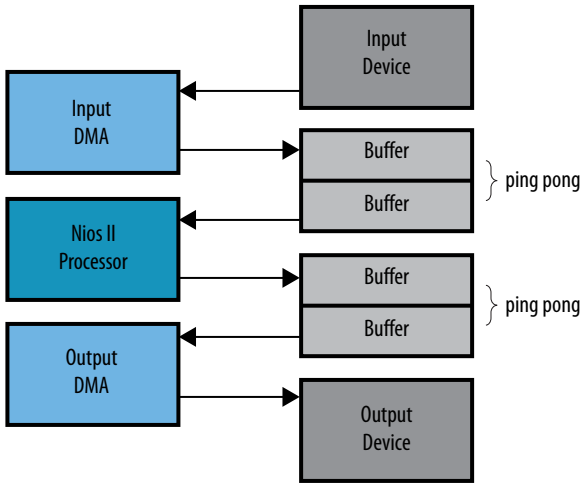
This section describes the optimization of memory partitioning in a video processing application to illustrate the concepts discussed earlier.

Application Description

This video processing application employs an algorithm that operates on a full frame of video data, line by line. Other details of the algorithm do not impact design of the memory subsystem. The data flow includes the following steps:

1. A dedicated DMA engine copies the input data from the video source to a buffer.
2. A Nios II processor operates on that buffer, performing the video processing algorithm and writing the result to another buffer.
3. A second dedicated DMA engine copies the output from the processor result buffer to the video output device.
4. The two DMAs provide an element of concurrency by copying input data to the next input buffer, and copying output data from the previous output buffer at the same time the processor is processing the current buffer, a technique commonly called ping-ponging.

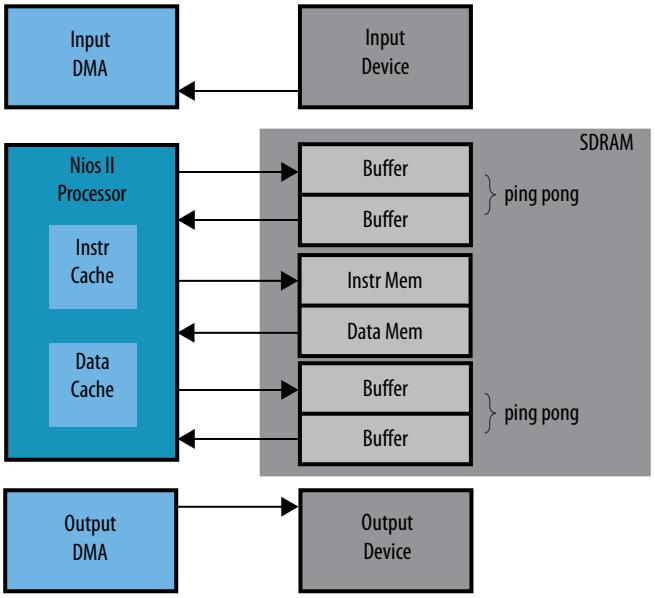
Figure 3-21: Sample Application Architecture



Initial Memory Partitioning

As a starting point, the application uses SDRAM for all of its storage and buffering, a commonly used memory architecture. The input DMA copies data from the video source to an input buffer in SDRAM. The Nios II processor reads from the SDRAM input buffer, processes the data, and writes the result to an output buffer, also located in SDRAM. In addition, the processor uses SDRAM for both its instruction and data memory, as shown below.

Figure 3-22: All Memory Implemented in SDRAM



Functionally, there is nothing wrong with this implementation. It is a frequently used, traditional type of embedded system architecture. It is also relatively inexpensive, because it uses only one external memory device; however, it is somewhat inefficient, particularly regarding its use of SDRAM. As the figure above illustrates, six different channels of data are accessed in the SDRAM.

- Processor instruction channel
- Processor data channel
- Input data from DMA
- Input data to processor
- Output data from processor
- Output data to DMA

With this many channels moving in and out of SDRAM simultaneously, especially at the high data rates required by video applications, the SDRAM bandwidth is easily the most significant performance bottleneck in the design.

Optimized Memory Partitioning

This design can be optimized to operate more efficiently. These optimizations are described in the following sections.

Add an External SRAM for Input Buffers

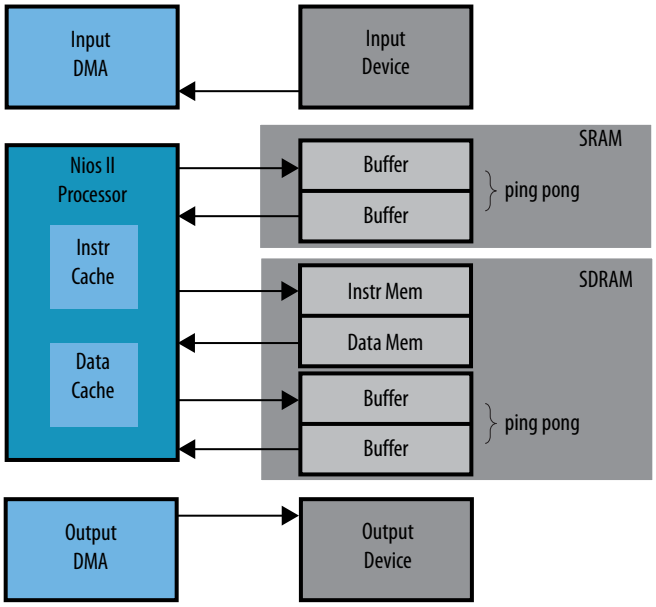
The first optimization to improve efficiency is to move the input buffering from the SDRAM to an external SRAM device. This technique creates performance gains for the following three reasons:

- The input side of the application achieves higher throughput because it now uses its own dedicated external SRAM to bring in video data.
- Two of the high-bandwidth channels from the SDRAM are eliminated, allowing the remaining SDRAM channels to achieve higher throughput.
- Eliminating two channels reduces the number of accesses to the SDRAM memory, leading to fewer SDRAM row changes, leading to higher throughput.

The redesigned system processes data faster, at the expense of more complexity and higher cost. The figure below illustrates the redesigned system.

If the video frames are small enough to fit in FPGA on-chip memory, you can use on-chip memory for the input buffers, saving the expense and complexity of adding an external SRAM device.

Figure 3-23: Input Channel Moved to External SSRAM



Note that four channels remain connected to SDRAM:

1. Processor instruction channel
2. Processor data channel
3. Output data from processor
4. Output data to DMA

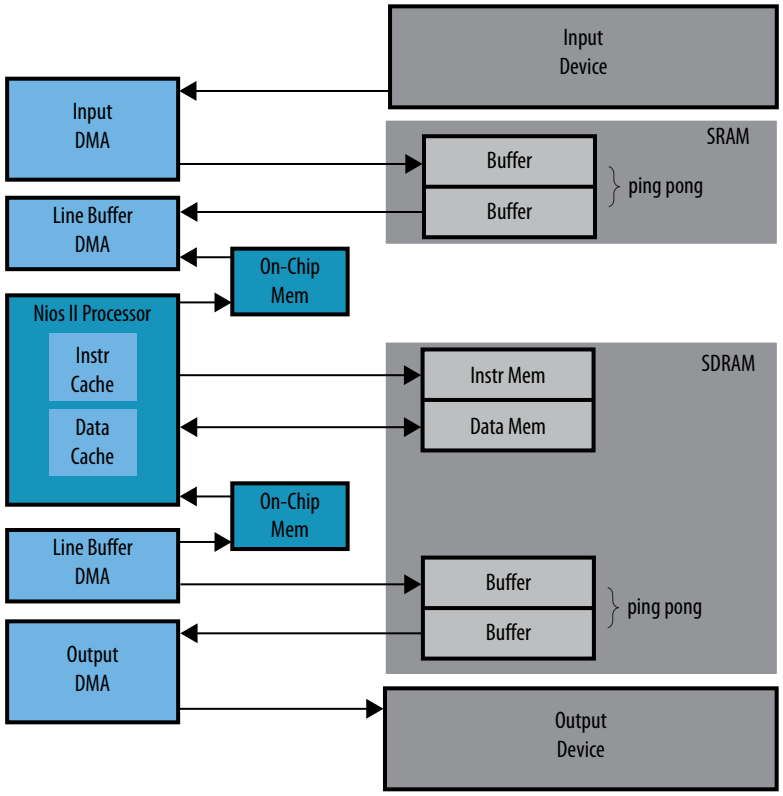
While we could probably achieve some additional performance benefit by adding a second external SRAM for the output channel, the benefit is not likely to be significant enough to outweigh the added cost and complexity. The reason is that only two of the four remaining channels require significant bandwidth from the SDRAM, the two video output channels. Assuming our Nios II processor contains both instruction and data caches, the SDRAM bandwidth required by the processor is likely to be relatively small. Therefore, sharing the SDRAM for processor instructions, processor data, and the video output channel is probably acceptable. If necessary, increasing the processor cache sizes can further reduce the processor's reliance on SDRAM bandwidth.

Add On-Chip Memory for Video Line Buffers

The final optimization is to add small on-chip memory buffers for input and output video lines. Because the processing algorithm operates on the video input one line at a time, buffering entire lines of input data in an on-chip memory improves performance. This buffering enables the Nios II processor to read all its input data from on-chip RAM—the fastest, lowest latency type of memory available.

The DMA fills these buffers ahead of the Nios II processor in a ping-pong scheme, in a manner analogous to the input frame buffers used for the external SRAM. The same on-chip memory line buffering scheme is used for processor output. The Nios II processor writes its output data to an on-chip memory line buffer, which is copied to the output frame buffer by a DMA after both the input and output ping-pong buffers flip, and the processor begins processing the next line. The figure below illustrates this memory architecture.

Figure 3-24: On-Chip Memories Added As Line Buffers



Document Revision History

Table 3-7: Hardware System Design with Quartus Prime and Qsys Chapter Revision History

Date	Version	Changes
December 2016	2016.12.19	Initial release.

Software System Design with a Nios II Processor

4

2016.12.19

ED_HANDBOOK

This chapter describes the software flow in a Nios II processor system design. It includes a detailed explanation and example on the Nios II command line tools that are provided in the Nios II Embedded Design Suite (EDS). Descriptions of both the Altera tools and the GNU tools are also included. Most of the commands are located in the `bon` and `sdk` subdirectories of your EDS installation.

Included is a description on how to develop the software flow and the software tools you can use in developing your embedded design system. Information about development with HAL drivers is also in this chapter.

Nios II Command-Line Tools

The Altera command line tools are useful for a range of activities, from board and system-level debugging to programming an FPGA configuration file (**.sof**). For these tools, the examples expand on the brief descriptions of the Altera-provided command-line tools for developing Nios II programs in "Altera-Provided Embedded Development Tools" in the Nios II Software Build Tools chapter of the *Nios II Gen2 Software Developer's Handbook*. The Nios II GCC toolchain contains the GNU Compiler Collection, GNU Binary Utilities (binutils), and newlib C library.

All of the commands described in this section are available in the Nios II command shell. For most of the commands, you can obtain help in this shell by typing:

```
<command name> --help
```

To start the Nios II command shell on Windows platforms, on the Start menu, click All Programs. On the **All Programs** menu, on the Altera submenu, on the Nios II EDS <*version*> submenu, click **Nios II** <*version*> **Command Shell**.

On Linux platforms, type the following command:

```
<Nios II EDS install path>/nios2_command_shell.shr
```

The command shell is a Bourne-again shell (bash) with a pre-configured environment.

Related Information

Nios II Software Build Tools

Altera Command-Line Tools for Board Bringup and Diagnostics

This section describes Altera command-line tools useful for Nios development board bringup and debugging.

jtagconfig

This command returns information about the devices connected to your host PC through the JTAG interface, for your use in debugging or programming. Use this command to determine if you configured your FPGA correctly.

Many of the other commands depend on successful JTAG connection. If you are unable to use other commands, check whether your JTAG chain differs from the simple, single-device chain used as an example in this section.

Type `jtagconfig --help` from a Nios II command shell to display a list of options and a brief usage statement.

jtagconfig Usage Example

To use the `jtagconfig` command, perform the following steps:

1. Open a Nios II command shell.
2. In the command shell, type the following command:

```
jtagconfig -n
```

Example 4-1: jtagconfig Example Response

```
$ jtagconfig -n
1) USB-Blaster [USB-0]
020050DD EP1S40/_HARDCOPY_FPGA_PROTOTYPE
Node 11104600
Node 0C006E00
```

The information in the response varies, depending on the particular FPGA, its configuration, and the JTAG connection cable type. The table below describes the information that appears in the response in the example.

Table 4-1: Interpretation of jtagconfig Command Response

Value	Description
`USB-Blaster [USB-0]`	The type of cable. You can have multiple cables connected to your workstation.
`EP1S40/_HARDCOPY_FPGA_PROTOTYPE`	The device name, as identified by silicon identification number.
`Node 11104600`	The node number of a JTAG node inside the FPGA. The appearance of a node number between 11104600 and 11046FF, inclusive, in this system's response confirms that you have a Nios II processor with a JTAG debug module.

Value	Description
`Note 0C006E00`	The node number of a JTAG node inside the FPGA. The appearance of a node number between 0C006E00 and 0C006EFF, inclusive, in this system's response confirms that you have a JTAG UART component.

The device name is read from the text file pgm_parts.txt in your Quartus Prime installation. In the example above, the name is EP1S40/_HARDCOPY_FPGA_PROTOTYPE because the silicon identification number on the JTAG chain for the FPGA device is 020050DD, which maps to the names EP1S40<*device-specific name*>, a couple of which end in the string _HARDCOPY_FPGA_PROTOTYPE. The internal nodes are nodes on the system-level debug (SLD) hub. All JTAG communication to an Altera FPGA passes through this hub, including advanced debugging capabilities such as the SignalTap II embedded logic analyzer and the debugging capabilities in the Nios II EDS.

The example above illustrates a single cable connected to a single-device JTAG chain. However, your computer can have multiple JTAG cables, connected to different systems. Each of these systems can have multiple devices in its JTAG chain. Each device can have multiple JTAG debug modules, JTAG UART modules, and other kinds of JTAG nodes. Use the `jtagconfig -n` command to help you understand the devices with JTAG connections to your host PC and how you can access them.

nios2-configure-sof

This command downloads the specified .sof and configures the FPGA according to its contents. At a Nios II command shell prompt, type `nios2-configure-sof --help` for a list of available command-line options.

You must specify the cable and device when you have more than one JTAG cable (USB-Blaster or ByteBlaster™ cable) connected to your computer or when you have more than one device (FPGA) in your JTAG chain. Use the `--cable` and `--device` options for this purpose.

nios2-configure-sof Usage Example

To use the `nios2-configure-sof` command, perform the following steps:

1. Open a Nios II command shell.
2. In the command shell, change to the directory in which your **.sof** is located. By default, the correct location is the top-level Quartus Prime project directory.
3. In the command shell, type the following command:

```
nios2-configure-sof
```

The Nios II EDS searches the current directory for a **.sof** and programs it through the specified JTAG cable.

system-console

The system-console command starts a Tcl-based command shell that supports low-level JTAG chain verification and full system-level validation.This tool is available in the Nios II EDS starting in version 8.0.

This application is very helpful for low-level system debug, especially when bringing up a system. It provides a Tcl-based scripting environment and many features for testing your system.

The following important command-line options are available for the `system-console` command:

- The `--script=<your script>.tcl` option directs the System Console to run your Tcl script.
- The `--cli` option directs the System Console to open in your existing shell, rather than opening a new window.
- The `--debug` option directs the System Console to redirect additional debug output to **stderr**.
- The `--project-dir=<project dir>` option directs the System Console to the location of your hardware project. Ensure that you're working with the project you intend—the JTAG chain details and other information depend on the specific project.
- The `--jdi=<JDI file>` option specifies the name-to-node mapping for the JTAG chain elements in your project.

For System Console usage examples and a comprehensive list of system console commands, refer to Analyzing and Debugging Designs with the System Console in volume 3 of the *Quartus Prime Handbook*, on-line training is available.

Related Information

- **Analyzing and Debugging Designs with System Console**
- **System Console Online Course**

Altera Command-Line Tools for Flash Programming

This section describes the command-line tools for programming your Nios II-based design in flash memory.

When you use the Nios II EDS to program flash memory, the Nios II EDS generates a shell script that contains the flash conversion commands and the programming commands. You can use this script as the basis for developing your own command-line flash programming flow.

For more details about the Nios II EDS and command-line usage of the Nios II Flash Programmer and related tools, refer to the *Nios II Flash Programmer User Guide*.

Related Information

Nios II Flash Programmer User Guide

nios2-flash-programmer

This command programs common flash interface (CFI) memory. Because the Nios II flash programmer uses the JTAG interface, the `nios2-flash-programmer` command has the same options for this interface as do other commands. You can obtain information about the command-line options for this command with the `--help` option.

Note: The `nios2-flash-programmer` has been replaced by the Quartus Prime Programmer flow for EPCS.

nios2-flash-programmer Usage Example

You can perform the following steps to program a CFI device:

1. Follow the steps in **nios2-download** on page 4-7, or use the Nios II EDS, to program your FPGA with a design that interfaces successfully to your CFI device.
2. Type the following command to verify that your flash device is detected correctly:

```
nios2-flash-programmer –debug –base=<base address>
```

where <*base address*> is the base address of your flash device. The base address of each component is displayed in Qsys. If the flash device is detected, the flash memory's CFI table contents are displayed.

3. Convert your file to flash format (**.flash**) using one of the utilities `elf2flash`, `bin2flash`, or `sof2flash` described in **elf2flash, bin2flash, and sof2flash** on page 4-5.
4. Type the following command to program the resulting .flash file in the CFI device:

```
nios2-flash-programmer –base=<base address> <file>.flashr
```

5. Optionally, type the following command to reset and start the processor at its reset address:

```
nios2-download –g –r
```

elf2flash, bin2flash, and sof2flash

These three commands are often used with the `nios2-flash-programmer` command. The resulting **.flash** file is a standard **.srec** file.

The following two important command-line options are available for the `elf2flash` command:

- The `-boot=<boot copier file>.srec` option directs the elf2flash command to prepend a bootloader S-record file to the converted ELF file.
- The `-after=<flash file>.flash` option places the generated **.flash** file—the converted ELF file—immediately following the specified **.flash** file in flash memory.

The -after option is commonly used to place the **.elf** file immediately following the **.sof** in an erasable, programmable, configurable serial EPCS or EPCQ flash device.

Caution: If you use an EPCS or EPCQ device, you must program the hardware image in the device before you program the software image. If you disregard this rule your software image will be corrupted.

Before it writes to any flash device, the Nios II flash programmer erases the entire sector to which it expects to write. In EPCS and EPCQ devices, however, if you generate the software image using the `elf2flash -after` option, the Nios II flash programmer places the software image directly following the hardware image, not on the next flash sector boundary. Therefore, in this case, the Nios II flash programmer does not erase the current sector before placing the software image. However, it does erase the current sector before placing the hardware image.

When you use the flash programmer through the Nios II SBT, you automatically create a script that contains some of these commands. Running the flash programmer creates a shell script (**.sh**) in the **Debug** or **Release** target directory of your project. This script contains the detailed command steps you used to program your flash memory.

Example 4-2: Sample Auto-Generated Script:

```
#!/bin/sh
#
# This file was automatically generated by the Nios II SBT For Eclipse.
#
# It will be overwritten when the flash programmer options change.
#

cd <full path to your project>/Debug

# Creating .flash file for the FPGA configuration
#"<Nios II EDS install path>/bin/sof2flash" --offset=0x400000 \
    --input="full path to your SOF" \
    --output="<your design>.flash"
```

```
# Programming flash with the FPGA configuration
#"<Nios II EDS install path>/bin/nios2-flash-programmer" --base=0x00000000 \
    --sidp=0x00810828 --id=1436046714 \
    --timestamp=1169569475 --instance=0 "<your design>.flash"
#
# Creating .flash file for the project
"<Nios II EDS install path>/bin/elf2flash" --base=0x00000000 --end=0x7fffff \
    --reset=0x0 \
    --input="<your project name>.elf" --output="ext_flash.flash" \
    --boot="<path to the bootloader>/boot_loader_cfi.srec"

# Programming flash with the project
"<Nios II EDS install path>/bin/nios2-flash-programmer" --base=0x00000000 \
    --sidp=0x00810828 --id=1436046714 \
    --timestamp=1169569475 --instance=0 "ext_flash.flash"

# Creating .flash file for the read only zip file system
"<Nios II EDS install path>/bin/bin2flash" --base=0x00000000 --
location=0x100000\
    --input="<full path to your binary file>" --output="<filename>.flash"

# Programming flash with the read only zip file system
"<Nios II EDS install path>/bin/nios2-flash-programmer" --base=0x00000000 \
    --sidp=0x00810828 --id=1436046714 \
    --timestamp=1169569475 --instance=0 "<filename>.flash"
```

The paths, file names, and addresses in the auto-generated script change depending on the names and locations of the files that are converted and on the configuration of your hardware design.

bin2flash Usage Example

To program an arbitrary binary file to flash memory, perform the following steps:

1. Type the following command to generate your **.flash** file:

```
bin2flash --location=<offset from the base address> \
    -input=<your file> --output=<your file>.flash
```

2. Type the following command to program your newly created file to flash memory:

```
nios2-flash-programmer -base=<base address> <your file>.flash
```

Altera Command-Line Tools for Software Development and Debug

This section describes Altera command-line tools that are useful for software development and debugging.

nios2-terminal

This command establishes contact with **stdin**, **stdout**, and **stderr** in a Nios II processor subsystem. stdin, stdout, and stderr are routed through a UART (standard UART or JTAG UART) module within this system.

The `nios2-terminal` command allows you to monitor **stdout**, **stderr**, or both, and to provide input to a Nios II processor subsystem through **stdin**. This command behaves the same as the `nios2-configure-sof` command described in **nios2-configure-sof** on page 4-3 with respect to JTAG cables and devices. However, because multiple JTAG UART modules may exist in your system, the `nios2-terminal` command requires explicit direction to apply to the correct JTAG UART module instance. Specify the instance using the -instance command-line option. The first instance in your design is 0 (`-instance "0"`). Additional instances are numbered incrementally, starting at 1 (`-instance "1"`).

nios2-download

This command parses Nios II **.elf** files, downloads them to a functioning Nios II processor, and optionally runs the .elf file.

As for other commands, you can obtain command-line option information with the `--help` option. The `nios2-download` command has the same options as the `nios2-terminal` command for dealing with multiple JTAG cables and Nios II processor subsystems.

nios2-download Usage Example

To download (and run) a Nios II **.elf** program:

1. Open a Nios II command shell.
2. Change to the directory in which your **.elf** file is located. If you use the Nios II SBT for development, the correct location is often the **Debug** or **Release** subdirectory of your top-level project.If you use the Nios II SBT, the correct location is the **app** folder.
3. In the command shell, type the following command to download and start your program:

```
nios2-download -g <project name>.elf
```

4. Optionally, use the `nios2-terminal` command to connect to view any output or provide any input to the running program.

nios2-stackreport

This command returns a brief report on the amount of memory still available for stack and heap from your project's **.elf** file.

This command does not help you to determine the amount of stack or heap space your code consumes during runtime, but it does tell you how much space your code has to work in.

Example 4-3: nios2-stackreport Command and Response

```
$ nios2-stackreport <your project>.elf
Info: (<your project>.elf) 6312 KBytes program size (code + initialized
data).
Info:                10070 KBytes free for stack + heap.
```

nios2-stackreport Usage Example

To use the `nios2-stackreport` command, perform the following steps:

1. Open a Nios II command shell.
2. Change to the directory in which your **.elf** file is located.
3. In the command shell, type the following command:

```
nios2-stackreport <your project>.elf
```

validate_zip

The Nios II EDS uses this command to validate that the files you use for the Read Only Zip Filing System are uncompressed. You can use it for the same purpose.

validate_zip Usage Example

To use the `validate_zip` command, perform the following steps:

1. Open a Nios II command shell.
2. Change to the directory in which your **.zip** file is located.
3. In the command shell, type the following command:

```
validate_zip <file>.zip
```

If no response appears, your **.zip** file is not compressed.

nios2-gdb-server

This command starts a GNU Debugger (GDB) JTAG conduit that listens on a specified TCP port for a connection from a GDB client, such as a `nios2-elf-gdb` client.

Occasionally, you may have to terminate a GDB server session. If you no longer have access to the Nios II command shell session in which you started a GDB server session, or if the offending GDB server process results from an errant Nios II SBT debugger session, you should stop the **nios2-gdb-server.exe** process on Windows platforms, or type the following command on Linux platforms:

```
pkill -9 -f nios2-gdb-server
```

nios2-gdb-server Usage Example

The Nios II SBT for Eclipse and most of the other available debuggers use the `nios2-gdb-server` and `nios2-elf-gdb` commands for debugging. You should never have to use these tools at this low level. However, in case you prefer to do so, this section includes instructions to start a GDB debugger session using these commands, and an example GDB debugging session.

You can perform the following steps to start a GDB debugger session:

1. Open a Nios II command shell.
2. In the command shell, type the following command to start the GDB server on the machine that is connected through a JTAG interface to the Nios II system you wish to debug:

```
nios2-gdb-server --tcpport 2342 --tcppersist
```

If the transfer control protocol port 2342 is already in use, use a different port.

Following is the system response:

```
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Pausing target processor: OK
Listening on port 2342 for connection from GDB:
```

Now you can connect to your server (locally or remotely) and start debugging.

3. Type the following command to start a GDB client that targets your **.elf** file:

```
nios2-elf-gdb <file>.elf
```

Example 4-4: Sample Debugging Session

```
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
```

```
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --target=nios2-elf"...
(gdb) target remote <your_host>:2342
Remote debugging using <your_host>:2342
OS_TaskIdle (p_arg=0x0) at sys/alt_irq.h:127
127 {
(gdb) load
Loading section .exceptions, size 0x1b0 lma 0x1000020
Loading section .text, size 0x3e4f4 lma 0x10001d0
Loading section .rodata, size 0x4328 lma 0x103e6c4
Loading section .rwdata, size 0x2020 lma 0x10429ec
Start address 0x10001d0, load size 281068
Transfer rate: 562136 bits/sec, 510 bytes/write.
(gdb) step
.
.
.
(gdb) quit
```

Possible commands include the standard debugger commands `load`, `step`, `continue`, `run`, and `quit`. Press Ctrl+c to terminate your GDB server session.

Altera Command-Line Nios II Software Build Tools

The Nios II software build tools are command-line utilities available from a Nios II command shell that enable you to create application, board support package (BSP), and library software for a particular Nios II hardware system. Use these tools to create a portable, self-contained makefile-based project that can be easily modified later to suit your build flow.

Unlike the Nios II SBT-based flow, proficient use of these tools requires some expertise with the GNU make-based software build flow. Before you use these tools, refer to the Nios II Software Build Tools and the Nios II Software Build Tools Reference chapters of the *Nios II Software Developer's Handbook*.

The following sections summarize the commands available for generating a BSP for your hardware design and for generating your application software. Many additional options are available in the Nios II software build tools. For an overview of the tools summarized in this section, refer to the Nios II Software Build Tools chapter of the *Nios II Software Developer's Handbook*.

Related Information

- **Nios II Software Build Tools**
- **Nios II Software Build Tools Reference**
- **Developing Nios II Software** on page 4-18

BSP Related Tools

Use the following command-line tools to create a BSP for your hardware design:

- `nios2-bsp-create-settings` creates a BSP settings file.
- `nios2-bsp-update-settings` updates a BSP settings file.
- `nios2-bsp-query-settings` queries an existing BSP settings file.
- `nios2-bsp-generate-files` generates all the files related to a given BSP settings file.
- `nios2-bsp` is a script that includes most of the functionality of the preceding commands.
- `create-this-bsp` is a high-level script that creates a BSP for a specific hardware design example.

Application Related Tools

Use the following commands to create and manipulate Nios II application and library projects:

- nios2-app-generate-makefile creates a makefile for your application.
- nios2-lib-generate-makefile creates a makefile for your application library.
- create-this-app is a high-level script that creates an application for a specific hardware design example.

Rebuilding Software from the Command Line

Rebuilding software after minor source code edits does not require a GUI. You can rebuild the project from a Nios II Command Shell, using your application's makefile. To build or rebuild your software, perform the following steps:

1. Open a Nios II Command Shell by executing one of the following steps, depending on your environment:
   - In the Windows operating system, on the Start menu, point to `Programs > Altera > Nios II EDS`, and click **Nios II Command Shell**.
   - In the Linux operating system, in a command shell, type the following sequence of commands:

     ```
     cd <Nios II EDS install path>
     ./nios2_command_shell.sh
     ```
2. Change to the directory in which your makefile is located. If you use the Nios II SBT for development, the correct location is often the **Debug** or **Release** subdirectory of your software project directory.
3. In the Command Shell, type one of the following commands:

   ```
   make
   ```

   or

   ```
   make -s
   ```

The example below illustrates the output of the make command run on a sample system.

Example 4-5: Sample Output From make -s Command

```
$ make -s
Creating generated_app.mk...
Creating generated_all.mk...
Creating system.h...
Creating alt_sys_init.c...
Creating generated.sh...
Creating generated.gdb...
Creating generated.x...
Compiling src1.c...
Compiling src2.c...
Compiling src3.c...
Compiling src4.c...
Compiling src5.c...
Linking project_name.elf...
```

If you add new files to your project or make significant hardware changes, recreate the project with the original tool (the Nios II SBT). Recreating the project recreates the makefile for the new version of your system after the modifications.

GNU Command-Line Tools

The Nios II GCC toolchain contains the GNU Compiler Collection, the GNU binutils, and the newlib C library. You can follow links to detailed documentation from the Nios II EDS documentation launchpad found in your Nios II EDS distribution. To start the launchpad on Windows platforms, on the Start menu, click **All Programs**. On the All Programs menu, on the Altera submenu, on the Nios II EDS *<version>* submenu, click **Literature**. On Linux platforms, open `<Nios II EDS install dir>/documents/index.htm` in a web browser. In addition, more information about the GNU GCC toolchain is available on the online.

nios2-elf-addr2line

This command returns a source code line number for a specific memory address. The command is similar to but more specific than the `nios2-elf-objdump` command described in **nios2-elf-objdump** on page 4-17 and the `nios2-elf-nm` command described in **nios2-elf-nm** on page 4-16.

Use the

```
nios2-elf-addr2line
```

command to help validate code that should be stored at specific memory addresses. The example below illustrates its usage and results:

Example 4-6: nios2-elf-addr2line Utility Usage Example

```
$ nios2-elf-addr2line --exe=<your project>.elf 0x1000020
<Nios II EDS install path>/components/altera_nios2/HAL/src/
alt_exception_entry.S:99
```

nios2-elf-addr2line Usage Example

To use the `nios2-elf-addr2line` command, perform the following steps:

1. Open a Nios II command shell.
2. In the command shell, type the following command:

```
nios2-elf-addr2line <your project>.elf <your_address_0>,\
<your_address_1>,...,<your_address_n>
```

If your project file contains source code at this address, its line number appears.

nios2-elf-gdb

This command is a GDB client that provides a simple shell interface, with built-in commands and scripting capability. A typical use of this command is illustrated in the section **nios2-gdb-server** on page 4-8.

nios2-elf-readelf

Use this command to parse information from your project's **.elf** file. The command is useful when used with **grep**, **sed**, or **awk** to extract specific information from your **.elf** file.

nios2-elf-readelf Usage Example

To display information about all instances of a specific function name in your .elf file, perform the following steps:

1. Open a Nios II command shell.
2. In the command shell, type the following command:

```
nios2-elf-readelf -symbols <project>.elf | grep <function name>
```

Example 4-7: Search for the http_read_line Function Using nios2-elf-readelf

```
$ nios2-elf-readelf.exe –s my_file.elf | grep http_read_line
1106: 01001168 160 FUNC GLOBAL DEFAULT 3 http_read_line
```

Table 4-2: Interpretation of nios2-elf-readelf Command Response

Value	Description
1106	Symbol instance number
0100168	Memory address, in hexadecimal format
160	Size of this symbol, in bytes
FUNC	Type of this symbol (function)
GLOBAL	Binding (values: GLOBAL, LOCAL, and WEAK)
DEFAULT	Visibility (values: DEFAULT, INTERNAL, HIDDEN, and PROTECTED)
3	Index
http_read_line	Symbol name

You can obtain further information about the ELF file format online. Each of the ELF utilities has its own main page.

nios2-elf-ar

This command generates an archive (**.a**) file containing a library of object (**.o**) files. The Nios II SBT uses this command to archive the System Library project.

nios2-elf-ar Usage Example

To archive your object files using the `nios2-elf-ar` command, perform the following steps:

1. Open a Nios II command shell.
2. Change to the directory in which your object files are located.
3. In the command shell, type the following command:

```
nios2-elf-ar q <archive_name>.a <object files>
```

The example shows how to create an archive of all of the object files in your current directory. In the example, the q option directs the command to append each object file it finds to the end of the archive.

After the archive file is created, it can be distributed for others to use, and included as an argument in linker commands, in place of a long object file list.

Example 4-8: nios2-elf-ar Command Response

```
$ nios2-elf-ar q <archive_name>.a *.o
nios2-elf-ar: creating <archive_name>.a
```

Linker

Use the `nios2-elf-g++` command to link your object files and archives into the final executable format, ELF.

Linker Usage Example

To link your object files and archives into a .elf file, open a Nios II command shell and call `nios2-elf-g++` with appropriate arguments. The following example command line calls the linker:

```
nios2-elf-g++ -T'<linker script>' -msys-crt0='<crt0.o file>' \
-msys-lib=<system library> -L '<The path where your libraries reside>' \
-DALT_DEBUG -O0 -g -Wall -mhw-mul -mhw-mulx -mno-hw-div \
-o <your project>.elf <object files> -lm
```

The exact linker command line to link your executable may differ. When you build a project in the Nios II SBT, you can see the command line used to link your application. To turn on this option in the Nios II SBT, on the Window menu, click **Preferences**, select the **Nios II** tab, and enable **Show command lines when running make**. You can also force the command lines to display by running make without the -s option from a Nios II command shell.

Note: Altera recommends that you not use the native linker `nios2-elf-ld` to link your programs. For the Nios II processor, as for all softcore processors, the linking flow is complex. The `g++ (nios2-elf-g++)` command options simplify this flow. Most of the options are specified by the -m command-line option, but the options available depend on the processor choices you make.

nios2-elf-size

This command displays the total size of your program and its basic code sections.

nios2-elf-size Usage Example

To display the size information for your program, perform the following steps:

1. Open a Nios II command shell.
2. Change to the directory in which your .elf file is located.
3. In the command shell, type the following command:

```
nios2-elf-size <project>.elf
```

Example 4-9: nios2-elf-size Command Usage

```
$ nios2-elf-size my_project.elf
text data bss dec hex filename
272904 8224 6183420 6464548 62a424 my_project.elf
```

nios2-elf-strings

This command displays all the strings in a **.elf** file.

nios2-elf-strings Usage Example

The command has a single required argument:

```
nios2-elf-strings <project>.elf
```

nios2-elf-strip

This command strips all symbols from object files. All object files are supported, including ELF files, object files (**.o**) and archive files (**.a**).

nios2-elf-strip Usage Example

`nios2-elf-strip` <*options*> <*project*>`.elf`

nios2-elf-strip Usage Notes

The `nios2-elf-strip` command decreases the size of the **.elf** file.

This command is useful only if the Nios II processor is running an operating system that supports ELF natively.If ELF is the native executable format, the entire **.elf** file is stored in memory, and the size savings matter.If not, the file is parsed and the instructions and data stored directly in memory, without the symbols in any case.

Linux is one operating system that supports ELF natively; uClinux is another. uClinux uses the flat (FLT) executable format, which is translated directly from the ELF.

nios2-elf-gdbtui

This command starts a GDB session in which a terminal displays source code next to the typical GDB console.

The syntax for the `nios2-elf-gdbtui` command is identical to that for the `nios2-elf-gdb` command described in **nios2-elf-gdb** on page 4-11.

Two additional GDB user interfaces are available for use with the Nios II GDB Debugger. CGDB, a cursor-based GDB UI, is available at *sourceforge.net*. The Data Display Debugger (DDD) is highly recommended.

Related Information

www.sourceforge.net

nios2-elf-gprof

This command allows you to profile your Nios II system.

For details about this command and the Nios II EDS-based results GUI, refer to *AN 391: Profiling Nios II Systems*.

Related Information

AN391: Profiling Nios II Systems

nios2-elf-gcc and g++

These commands run the GNU C and C++ compiler, respectively, for the Nios II processor.

Compilation Command Usage Example

The following simple example shows a command line that runs the GNU C or C++ compiler:

```
nios2-elf-gcc(g++) <options> -o <object files> <C files>
```

More Complex Compilation Example

The example below is a Nios II EDS-generated command line that compiles C code in multiple files in many directories.

Example 4-10: Example nios2-elf-gcc Command Line

```
nios2-elf-gcc -xc -MD -c \
-DSYSTEM_BUS_WIDTH=32 -DALT_NO_C_PLUS_PLUS -DALT_NO_INSTRUCTION_EMULATION \
-DALT_USE_SMALL_DRIVERS -DALT_USE_DIRECT_DRIVERS -DALT_PROVIDE_GMON \
-I.. -I/cygdrive/c/Work/Projects/demo_reg32/Designs/std_2s60_ES/software/\
reg_32_example_0_syslib/Release/system_description \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_timer/HAL/inc \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_timer/inc \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_jtag_uart/HAL/
inc \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_jtag_uart/inc \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_pio/inc \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_lcd_16207/HAL/
inc \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_lcd_16207/inc \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_sysid/HAL/inc \
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_sysid/inc \
-I/cygdrive/c/altera/70_b31/nios2eds/components/altera_nios2/HAL/inc \
-I/cygdrive/c/altera/70_b31/nios2eds/components/altera_hal/HAL/inc \
-DALT_SINGLE_THREADED -D__hal__ -pipe -DALT_RELEASE -O2 -g -Wall\
-mhw-mul -mhw-mulx -mno-hw-div -o obj/reg_32_buttons.o ../reg_32_buttons.c
```

nios2-elf-c++filt

This command demangles C++ mangled names. C++ allows multiple functions to have the same name if their parameter lists differ; to keep track of each unique function, the compiler mangles, or decorates, function names. Each compiler mangles functions in a particular way.

For a full explanation, including more details about how the different compilers mangle C++ function names, refer to standard reference sources for the C++ language compilers.

nios2-elf-c++filt Usage Example

To display the original, demangled function name that corresponds to a particular symbol name, you can type the following command:

```
nios2-elf-c++filt -n <symbol name>
```

For example,

```
nios2-elf-c++filt -n _Z11my_functionv
```

More Complex nios2-elf-c++filt Example

The following example command line causes the display of all demangled function names in an entire file:

```
nios2-elf-strings <file>.elf | grep ^_Z | nios2-elf-c++filt -n
```

In this example, the `nios2-elf-strings` operation outputs all strings in the **.elf** file. This output is piped to a `grep` operation that identifies all strings beginning with `_Z`. (GCC always prepends mangled function names with `_Z`). The output of the `grep` command is piped to a `nios2-elf-c++filt` command. The result is a list of all demangled functions in a GCC C++ **.elf** file.

nios2-elf-nm

This command list the symbols in a **.elf** file.

nios2-elf-nm Usage Example

The following two simple examples illustrate the use of the nios2-elf-nm command:

- `nios2-elf-nm <project>.elf`
- `nios2-elf-nm <project>.elf | sort -n`

More Complex nios2-elf-nm Example

To generate a list of symbols from your .elf file in ascending address order, use the following command:

```
nios2-elf-nm <project>.elf | sort -n > <project>.elf.nm
```

The <*project*>**.elf.nm** file contains all of the symbols in your executable file, listed in ascending address order. In this example, the `nios2-elf-nm` command creates the symbol list. In this text list, each symbol's address is the first field in a new line. The `-n` option for the `sort` command specifies that the symbols be sorted by address in numerical order instead of the default alphabetical order.

nios2-elf-objcopy

Use this command to copy from one binary object format to another, optionally changing the binary data in the process.

Though typical usage converts from or to ELF files, the objcopy command is not restricted to conversions from or to ELF files. You can use this command to convert from, and to, any of the formats listed in the table below.

Table 4-3: -objcopy Binary Formats

Command (...-objcopy)	Comments
elf32-littlenios2, elf32-little	Header little endian, data little endian, the default and most commonly used format
elf32-bignios2, elf32-big	Header big endian, data big endian
srec	S-Record (SREC) output format
symbolsrec	SREC format with all symbols listed in the file header, preceding the SREC data
tekhex	Tektronix hexadecimal (TekHex) format

Command (...-objcopy)	Comments
binary	Raw binary format Useful for creating binary images for storage in flash on your embedded system
ihex	Altera hexadecimal (ihex) format

You can obtain information about the TekHex, ihex, and other text-based binary representation file formats online. As of the initial publication of this handbook, you can refer to the *sbprojects.com* knowledge-base entry on file formats.

Related Information

www.sbprojects.com

nios2-elf-objcopy Usage Example

To create an SREC file from an ELF file, use the following command:

```
nios2-elf-objcopy –O srec <project>.elf <project>.srec
```

ELF is the assumed binary format if none is listed. For information about how to specify a different binary format, in a Nios II command shell, type the following command:

```
nios2-elf-objcopy --help
```

nios2-elf-objdump

Use this command to display information about the object file, usually an ELF file.

The `nios2-elf-objdump` command supports all of the binary formats that the `nios2- elf-objcopy` command supports, but ELF is the only format that produces useful output for all command-line options.

nios2-elf-objdump Usage Description

The Nios II EDS uses the following command line to generate object dump files:

```
nios2-elf-objdump -D -S -x <project>.elf > <project>.elf.objdump
```

nios2-elf-ranlib

Calling `nios2-elf-ranlib` is equivalent to calling `nios2-elf-ar` with the `-s` option (`nios2-elf-ar -s`).

For further information about this command, refer to **nios2-elf-ar** on page 4-12 or type `nios2-elf-ar --help` in a Nios II command shell.

Developing Nios II Software

This section provides in-depth information about software development for the Altera Nios® II processor. It complements the *Nios II Gen2 Software Developer's Handbook* by providing the following additional information:

- **Recommended design practices**—Best practice information for Nios II software design, development, and deployment.
- **Implementation information**—Additional in-depth information about the implementation of application programming interfaces (APIs) and source code for each topic, if available.
- **Pointers to topics**—Informative background and resource information for each topic, if available.

Before reading this section, you should be familiar with the process of creating a simple board support package (BSP) and an application project using the Nios II Software Build Tools development flow. The Software Build Tools flow is supported by Nios II Software Build Tools for Eclipse™ as well as the Nios II Command Shell. This section focuses on the Nios II Software Build Tools for Eclipse, but most information is also applicable to project development in the Command Shell.

The following resources provide training on the Nios II SW Build Tools development flow:

- Online training demonstrations located on the Embedded Software Designer Curriculum page of the Altera website.
- Documentation located on the Documentation: Nios II Processor page of the Altera website, especially the "Getting Started from the Command Line" and "Getting Started with the Graphical User Interface" chapters of the *Nios II Gen2 Software Developer's Handbook*.
- Example designs provided with the Nios II Embedded Design Suite (EDS). The online training demonstrations describe these software design examples, which you can use as-is or as the basis for your own more complex designs.

This section is structured according to the Nios II software development process. Each section describes Altera's recommended design practices to accomplish a specific task.

When you install the Nios II EDS, it is installed in the same directory with the Quartus Prime software. For example, if the Quartus Prime software is installed on the Windows operating system, and the root directory of the Quartus Prime software is `c:\altera\<version>\quartus`, then the root directory of the Nios II EDS is `c:\altera\<version>\nios2eds`. For simplicity, this handbook refers to the **nios2eds** directory as:

```
<Nios II EDS install dir>
```

Related Information

- **Nios II Gen2 Software Developer's Handbook**
- **Embedded SW Designer Curriculum**
- **Documentation: Nios II Processor**

Software Development Cycle

The Nios II EDS includes a complete set of C/C++ software development tools for the Nios II processor. In addition, a set of third-party embedded software tools is provided with the Nios II EDS. This set includes the MicroC/OS-II real-time operating system and the NicheStack TCP/IP networking stack. This section focuses on the use of the Altera-created tools for Nios II software generation. It also includes some discussion of third-party tools.

The Nios II EDS is a collection of software generation, management, and deployment tools for the Nios II processor. The toolchain includes tools that perform low-level tasks and tools that perform higher-level tasks using the lower-level tools. For more information on Linux, refer to rocketboards.org.

Related Information

- **Altera System on a Programmable Chip (Qsys) Solutions** on page 3-5
- **Nios II Software Development Process** on page 4-21
- **rocketboards.org**

Nios II Software Design

This section contains brief descriptions of the software design tools provided by the Nios II EDS, including the Nios II SBT development flow.

Nios II Tools Overview

The Nios II EDS provides the following tools for software development:

- GNU toolchain: GCC-based compiler with the GNU binary utilities

  Note: For an overview of these and other Altera-provided utilities, refer to the "Nios II Command-Line Tools" chapter of this handbook.
- Nios II processor-specific port of the newlib C library
- Hardware abstraction layer (HAL)

  The HAL provides a simple device driver interface for programs to communicate with the underlying hardware. It provides many useful features such as a POSIX-like application program interface (API) and a virtual-device file system.

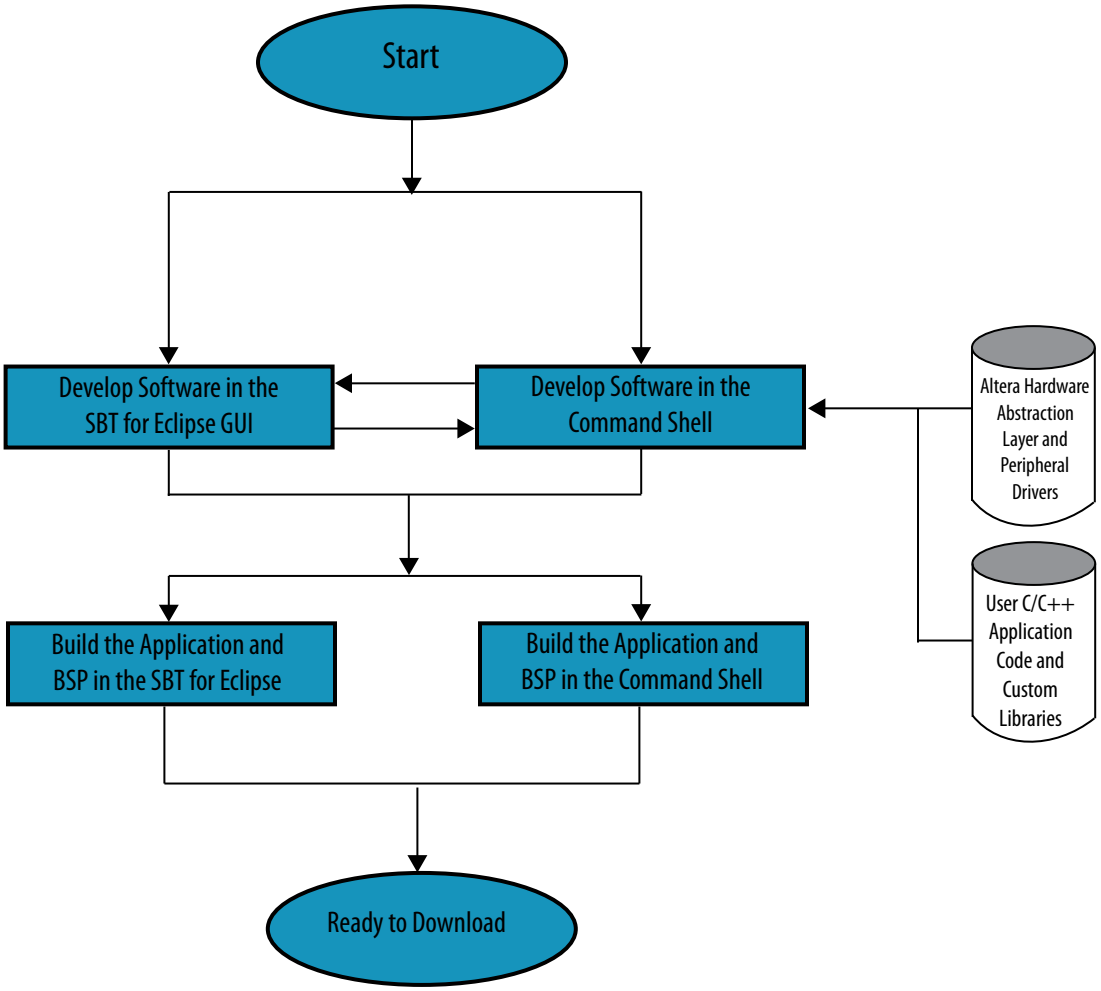
  For more information about the Altera HAL, refer to The Hardware Abstraction Layer section of the *Nios II Gen2 Software Developer's Handbook*.
- Nios II SBT

  The Nios II SBT development flow is a scriptable development flow. It includes the following user interfaces:

  - The Nios II SBT for Eclipse—a GUI that supports creating, modifying, building, running, and debugging Nios II programs. It is based on the Eclipse open development platform and Eclipse C/C++ development toolkit (CDT) plug-ins.
  - The Nios II SBT command-line interface—From this interface, you can execute SBT command utilities, and use scripts (or other tools) to combine the command utilities in many useful ways.

    For more information about the Nios II SBT flow, refer to the Developing Nios II Software chapter of this handbook.

Figure 4-1: Nios II Software Development Flows: Developing Software



Altera recommends that you view and begin your design with one of the available software examples that are installed with the Nios II EDS. From simple "Hello, World" programs to networking and RTOS-based software, these examples provide good reference points and starting points for your own software development projects. The Hello World Small example program illustrates how to reduce your code size without losing all of the conveniences of the HAL.

Note: Altera recommends that you use an Altera development kit or custom prototype board for software development and debugging. Many peripheral and system-level features are available only when your software runs on an actual board.

Related Information

- **Nios II Command-Line Tools** on page 4-1
- **The Hardware Abstraction Layer**
- **Developing Nios II Software** on page 4-18

Nios II Software Build Tools

The Nios II SBT flow uses the Software Build Tools to provide a flexible, portable, and scriptable software build environment. Altera recommends that you use this flow. The SBT includes a command-line environment and fits easily in your preferred software or system development environment.

The SBT flow requires that you have a **.sopcinfo** file for your system. The flow includes the following steps to create software for your system:

1. Create a board support package (BSP) for your system. The BSP is a layer of software that interacts with your development system. It is a makefile-based project.
2. Create your application software:
   a. Write your code.
   b. Generate a makefile-based project that contains your code.
3. Iterate through one or both of these steps until your design is complete.

For more information, refer to the software example designs that are shipped with every release of the Nios II EDS. For more information about these examples, refer to one of the following sections:

- "Getting Started" in the Getting Started with the Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*.
- "Nios II Example Design Scripts" in the Nios II Software Build Tools Reference chapter *of the Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Getting Started with the Graphical User Interface**
- **Nios II Software Build Tools Reference**

Nios II Software Development Process

This section provides an overview of the Nios II software development process and introduces terminology. The rest of the chapter elaborates the description in this section.

The Nios II software generation process includes the following stages and main hardware configuration tools:

1. Hardware configuration
   - Qsys
   - Quartus Prime software
2. Software project management
   - BSP configuration
   - Application project configuration
   - Editing and building the software project
   - Running, debugging, and communicating with the target
   - Ensuring hardware and software coherency
   - Project management
3. Software project development

- Developing with the Hardware Abstraction Layer (HAL)
- Programming the Nios II processor to access memory
- Writing exception handlers
- Optimizing the application for performance and size
- Real-time operating system (RTOS) support

4. Application deployment
   - Linking (run-time memory)
   - Boot loading the system application
   - Programming flash memory

In this list of stages and tools, the subtopics under the topics Software project management, Software project development, and Application deployment correspond closely to sections in the chapter.

You create the hardware for the system using the Quartus Prime and Qsys software. The main output produced by generating the hardware for the system is the SRAM Object File (**.sof**), which is the hardware image of the system, and the Qsys Information File (**.sopcinfo**), which describes the hardware components and connections.

Note: The key file required to generate the application software is the **.sopcinfo** file.

The software generation tools use the **.sopcinfo** file to create a BSP project. The BSP project is a collection of C source, header and initialization files, and a makefile for building a custom library for the hardware in the system. This custom library is the BSP library file (.a). The BSP library file is linked with your application project to create an executable binary file for your system, called an application image. The combination of the BSP project and your application project is called the software project.

The application project is your application C source and header files and a makefile that you can generate by running Altera-provided tools. You can edit these files and compile and link them with the BSP library file using the makefile. Your application sources can reference all resources provided by the BSP library file. The BSP library file contains services provided by the HAL, which your application sources can reference. After you build your application image, you can download it to the target system, and communicate with it through a terminal application.

You can access the makefile in the Eclipse **Project Explorer** view after you have created your project in the Nios II Software Build Tools for Eclipse framework.

The software project is flexible: you can regenerate it if the system hardware changes, or modify it to add or remove functionality, or tune it for your particular system. You can also modify the BSP library file to include additional Altera-supplied software packages, such as the read-only zip file system or TCP/IP networking stack (the NicheStack TCP/IP Stack). Both the BSP library file and the application project can be configured to build with different parameters, such as compiler optimizations and linker settings.

If you change the hardware system, you must recreate, update or regenerate the BSP project to keep the library header files up-to-date.

For information about how to keep your BSP up-to-date with your hardware, refer to "Revising Your BSP" in the Nios II Software Build Tools chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Revising Your BSP

Software Project Mechanics

This section describes the recommended ways to edit, build, download, run, and debug your software application, primarily using the Nios II Software Build Tools for Eclipse.

The Nios II Software Build Tools flow is the recommended design flow for hardware designs that contain a Nios II processor. This section describes how to configure BSP and application projects, and the process of developing a software project for a system that contains a Nios II processor, including ensuring coherency between the software and hardware designs.

Software Tools Background

The Nios II EDS provides a sophisticated set of software project generation tools to build your application image. The Nios II Software Build Tools flow is available for project creation. The Nios II Software Build Tools flow includes the Software Build Tools command-line interface and the Nios II Software Build Tools for Eclipse.

The Nios II Software Build Tools for Eclipse is the recommended flow. The Nios II Software Build Tools for Eclipse does not support the following Nios II Integrated Development Environment (IDE) feature:

- `stdio` output to an RS-232 UART cannot display on the System Console. To display stdio output on the System Console, configure your BSP to use a JTAG UART peripheral for stdout, using the `hal.stdout` BSP setting. If no JTAG UART is available in your hardware system, you can run **nios2-terminal** in a separate Nios II Command Shell to capture stdio output.

Altera recommends that you use the Nios II Software Build Tools for Eclipse to create new software projects. The Nios II Software Build Tools are the basis for Altera's future development.

A graphical user interface for configuring BSP libraries, called the Nios II BSP Editor, is also available. The BSP Editor is integrated with the Nios II Software Build Tools for Eclipse, and can also be used independently.

Development Flow Guidelines

The Nios II Software Build Tools flow provides many services and functions for your use. Until you become familiar with these services and functions, Altera recommends that you adhere to the following guidelines to simplify your development effort:

- **Begin with a known hardware design**—The All Design Examples page of the Altera website includes a set of known working designs, called hardware example designs, which are excellent starting points for your own design. In addition, the *Nios II Hardware Development Tutorial* walks through some example designs.
- **Begin with a known software example design**—The Nios II EDS includes a set of preconfigured application projects for you to use as the starting point of your own application. Use one of these designs and parameterize it to suit your application goals.
- **Follow pointers to documentation**—Many of the application and BSP project source files include inline comments that provide additional information.
- **Make incremental changes**—Regardless of your end-application goals, develop your software application by making incremental, testable changes, to compartmentalize your software development process. Altera recommends that you use a version control system to maintain distinct versions of your source files as you develop your project.

Related Information

- **All Design Examples**
- **Nios II Hardware Development Tutorial**

Nios II Software Build Tools

The Nios II Software Build Tools are a collection of command-line utilities and scripts. These tools allow you to build a BSP project and an application project to create an application image. The BSP project is a

parameterizable library, customized for the hardware capabilities and peripherals in your system. When you create a BSP library file from the BSP project, you create it with a specific set of parameter values. The application project consists of your application source files and the application makefile. The source files can reference services provided by the BSP library file.

For the full list of utilities and scripts in the Nios II Software Build Tools flow, refer to "Altera-Provided Embedded Development Tools" in the Nios II Software Build Tools chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Nios II Software Build Tools

The Nios II Software Build Tools for Eclipse

The Nios II Software Build Tools for Eclipse provide a consistent development platform that works for all Nios II processor systems. You can accomplish most software development tasks in the Nios II Software Build Tools for Eclipse, including creating, editing, building, running, debugging, and profiling programs.

The Nios II Software Build Tools for Eclipse are based on the popular Eclipse framework and the Eclipse C/C++ development toolkit (CDT) plug-ins. Simply put, the Nios II Software Build Tools for Eclipse provides a GUI that runs the Nios II Software Build Tools utilities and scripts behind the scenes.

For detailed information about the Nios II Software Build Tools for Eclipse, refer to the Getting Started with the Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*. For details about Eclipse, visit the Eclipse Foundation website.

Related Information

- **Getting Started with the Graphical User Interface**
- **www.eclipse.com**

The Nios II Software Build Tools Command Line

In the Nios II Software Build Tools command line development flow, you create, modify, build, and run Nios II programs with Nios II Software Build Tools commands typed at a command line or embedded in a script.

To debug your program, import your Software Build Tools projects to Eclipse. You can further edit, rebuild, run, and debug your imported project in Eclipse.

For further information about the Nios II Software Build Tools and the Nios II Command Shell, refer to the "Getting Started from the Command Line" chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Getting Started from the Command Line

Configuring BSP and Application Projects

This section describes some methods for configuring the BSP and application projects that comprise your software application, while encouraging you to begin your software development with a software example design.

For information about using version control, copying, moving and renaming a BSP project, and transferring a BSP project to another person, refer to "Common BSP Tasks" in the Nios II Software Build Tools chapter of the *Nios Gen2 II Software Developer's Handbook*.

Related Information

Nios II Software Build Tools

Software Example Designs

The best way to become acquainted with the Nios II Software Build Tools flow and begin developing software for the Nios II processor is to use one of the pre-existing software example designs that are provided with the Nios II EDS. The software example designs are preconfigured software applications that you can use as the basis for your own software development. The software examples can be found in the Nios II installation directory.

For more information about the software example designs provided in the Nios II EDS, refer to "Nios II Embedded Design Examples" in the Overview of Nios II Embedded Development chapter of the *Nios II Gen2 Software Developer's Handbook*.

To use a software example design, follow these steps:

1. Set up a working directory that contains your system hardware, including the system **.sopcinfo** file.

   Note: Ensure that you have compiled the system hardware with the Quartus Prime software to create up-to-date **.sof** and **.sopcinfo** files.
2. Start the Nios II Software Build Tools for Eclipse as follows:
   - In the Windows operating system, on the Start menu, point to **Programs > Altera > Nios II EDS** <*version*>, and click **Nios II** <*version*> **Software Build Tools for Eclipse**.
   - In the Linux operating system, in a command shell, type `eclipse-nios2`.
3. Right-click anywhere in the **Project Explorer** view, point to **New** and click **Nios II Application and BSP from Template**.
4. Select an appropriate software example from the **Templates** list.

   Note: You must ensure that your system hardware satisfies the requirements for the software example design listed under **Template description**. If you use an Altera Nios II development kit, the software example designs supplied with the kit are guaranteed to work with the hardware examples included with the kit.
5. Next to **Information File Name**, browse to your working directory and select the **.sopcinfo** file associated with your system.
6. In a multiprocessor design, you must select the processor on which to run the software project.

   Note: If your design contains a single Nios II processor, the processor name is automatically filled in.
7. Fill in the project name.
8. Click Next.
9. Select **Create a new BSP project based on the application project template**.
10. Click **Finish**. The Nios II Software Build Tools generate an Altera HAL BSP for you.

If you do not want the Software Build Tools for Eclipse to automatically create a BSP for you, at Step 9, select Select an existing BSP project from your workspace. You then have several options:

- You can import a pre-existing BSP by clicking **Import**.
- You can create a HAL or MicroC/OS-II BSP as follows:
  - Click **Create**. **The Nios II Board Support Package** dialog box appears.
  - Next to **Operating System**, select either **Altera HAL** or **Micrium MicroC/OS-II**.

    You can select the operating system only at the time you create the BSP. To change operating systems, you must create a new BSP.

Related Information

Overview of Nios II Embedded Development

Selecting the Operating System (HAL versus MicroC/OS-II RTOS)

You have a choice of the following run-time environments (operating systems) to incorporate in your BSP library file:

- The Nios II HAL—A lightweight, POSIX-like, single-threaded library, sufficient for many applications.
- The MicroC/OS-II RTOS—A real-time, multi-threaded environment. The Nios II implementation of MicroC/OS-II is based on the HAL, and includes all HAL services.

After you select HAL or MicroC/OS-II, you cannot change the operating system for this BSP project.

Configuring the BSP Project

The BSP project is a configurable library. You can configure your BSP project to incorporate your optimization preferences—size, speed, or other features—in the custom library you create. This custom library is the BSP library file (**.a**) that is used by the application project.

Creating the BSP project populates the target directory with the BSP library file source and build file scripts. Some of these files are copied from other directories and are not overwritten when you recreate the BSP project. Others are generated when you create the BSP project.

The most basic tool for configuring BSPs is the BSP setting. Throughout this section, many of the project modifications you can make are based on BSP settings. In each case, this section presents the names of the relevant settings, and explains how to select the correct setting value. You can control the value of BSP settings several ways: on the command line, with a Tcl script, by directly adjusting the settings with the BSP Editor, or by importing a Tcl script to the BSP Editor.

Another powerful tool for configuring a BSP is the software package. Software packages add complex capabilities to your BSP. As when you work with BSP settings, you can add and remove software packages on the command line, with a Tcl script, directly with the BSP Editor, or by importing a Tcl script to the BSP Editor.

Altera recommends that you use the Nios II BSP Editor to configure your BSP project. To start the Nios II BSP Editor from the Nios II Software Build Tools for Eclipse, right-click an existing BSP, point to **Nios II**, and click **BSP Editor**.

For detailed information about how to manipulate BSP settings and add and remove software packages with the BSP Editor, refer to "Using the BSP Editor" in the Getting Started with the Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*. This chapter also discusses how to use Tcl scripts in the BSP Editor.

For information about manipulating BSP settings and controlling software packages at the command line, refer to "Nios II Software Build Tools Utilities" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

For details about available BSP settings, refer to "Settings Managed by the Software Build Tools" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

For a discussion of Tcl scripting, refer to "Software Build Tools Tcl Commands" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Note: Do not edit BSP files, because they are overwritten by the Software Build Tools the next time the BSP is generated.

Related Information

- **Nios II Software Build Tools**
- **Getting Started with the Graphical User Interface**
- **Nios II Software Build Tools Reference**

MicroC/OS-II RTOS Configuration Tips

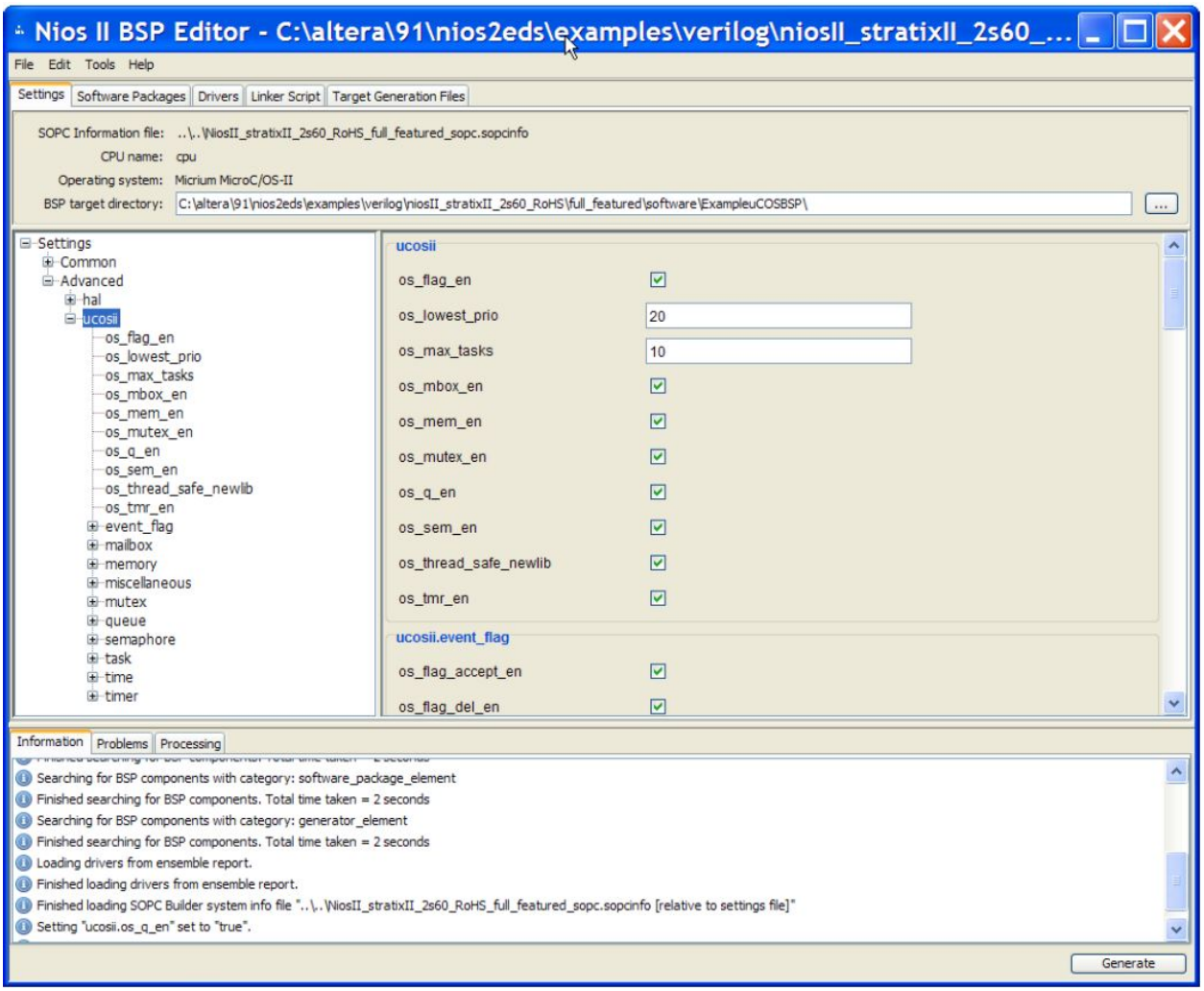
If you use the MicroC/OS-II RTOS environment, be aware of the following properties of this environment:

- **MicroC/OS-II BSP settings**—The MicroC/OS-II RTOS supports many configuration options. All of these options can be enabled and disabled with BSP settings. Some of the options are enabled by default. A comprehensive list of BSP settings for MicroC/OS-II is shown in the **Settings** tab of the Nios II BSP Editor.

  Note: The MicroC/OS-II BSP settings are also described in "Settings Managed by the Software Build Tools" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.
- MicroC/OS-II setting modification—Modifying the MicroC/OS-II options modifies the system.h file, which is used to compile the BSP library file.
- MicroC/OS-II initialization—The core MicroC/OS-II RTOS is initialized during the execution of the C run-time initialization (crt0) code block. After the crt0 code block runs, the MicroC/OS-II RTOS resources are available for your application to use. For more information, refer to "crt0 Initialization".

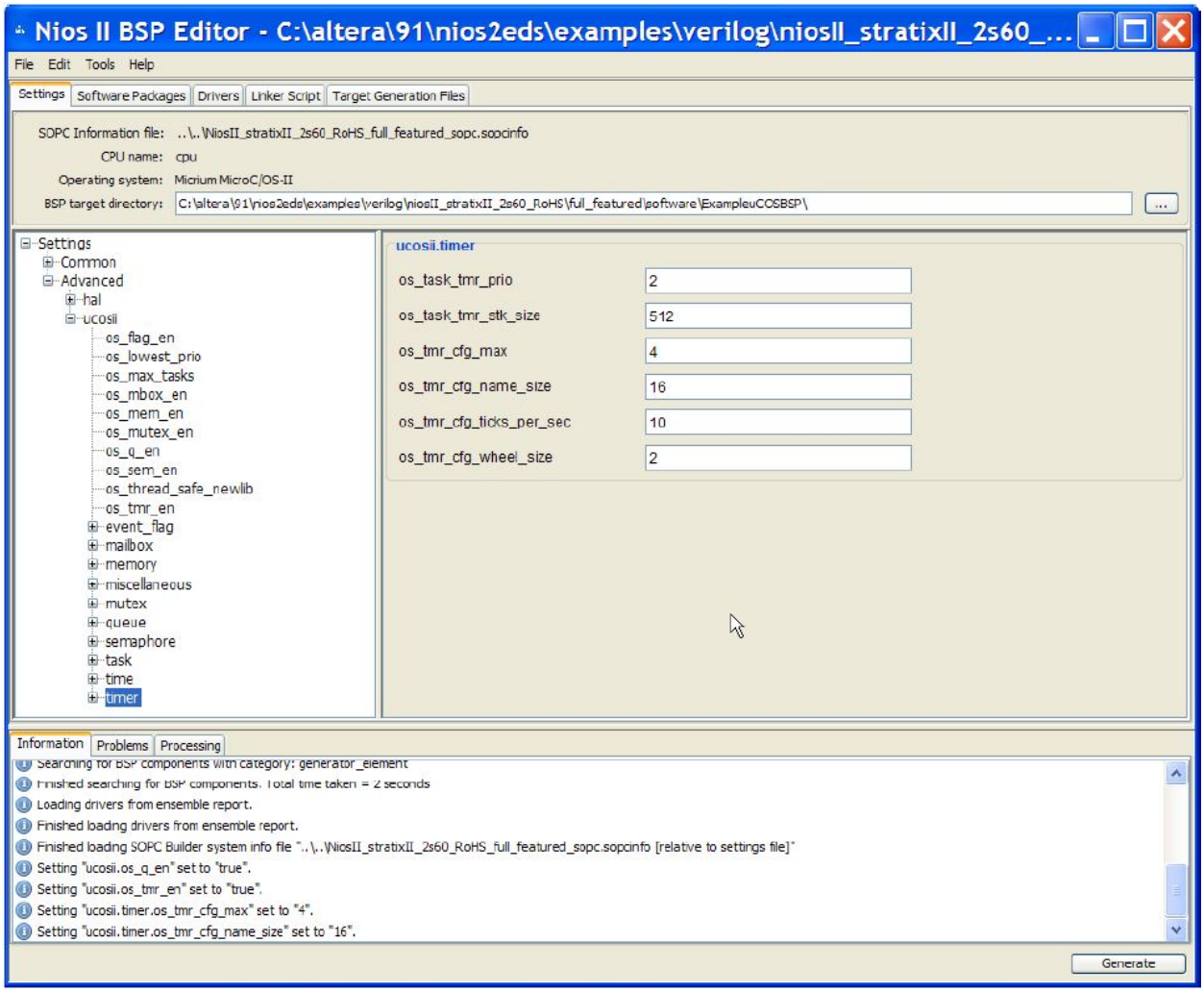
You can configure MicroC/OS-II with the BSP Editor. Figure 2–1 shows how you enable the MicroC/OS-II timer and queue code. The figure below shows how you specify a maximum of four timers for use with MicroC/OS-II.

Figure 4-2: Enabling MicroC/OS-II Timers and Queues in BSP Editor



The MicroC/OS-II configuration script in the example below performs the same MicroC/OS-II configuration as in the figure above and **Figure 4-3**: it enables the timer and queue code, and specifies a maximum of four timers.

Figure 4-3: Configuring MicroC/OS-II for Four Timers in BSP Editor



Example 4-11: MicroC/OS-II Tcl Configuration Script Example (ucosii_conf.tcl)

```
#enable code for UCOSII timers
set_setting ucosii.os_tmr_en 1

#enable a maximum of 4 UCOSII timers
set_setting ucosii.timer.os_tmr_cfg_max 4

#enable code for UCOSII queues
set_setting ucosii.os_q_en 1
```

Related Information

- **Nios II Software Build Tools Reference**
- **crt0 Initialization** on page 4-42

HAL Configuration Tips

If you use the HAL environment, be aware of the following properties of this environment:

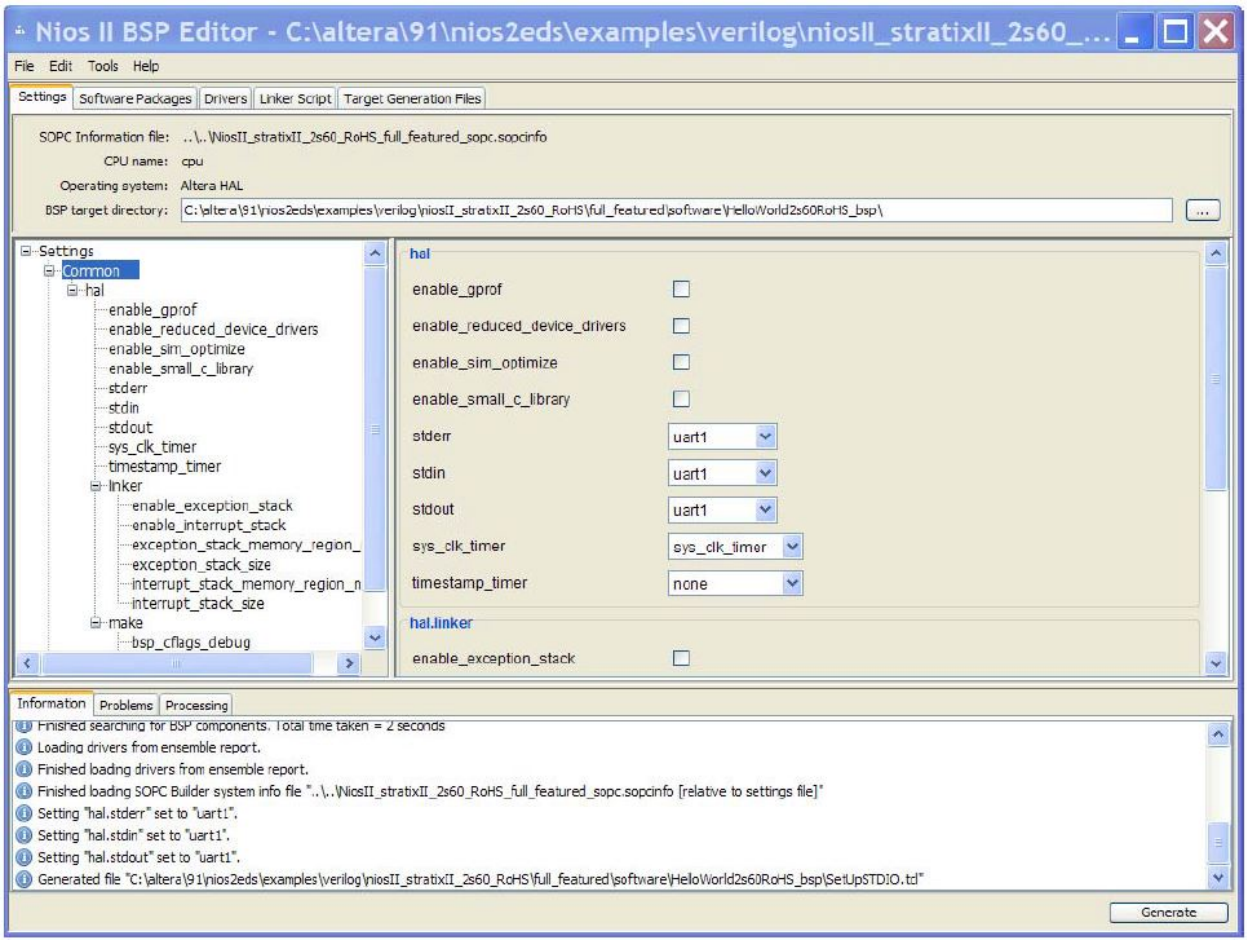
- **HAL BSP settings**—A comprehensive list of options is shown in the Settings tab in the Nios II BSP Editor. These options include settings to specify a pre- and post-process to run for each C or C++ file compiled, and for each file assembled or archived.

  Note: For more information about BSP settings, refer to "Settings Managed by the Software Build Tools" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.
- **HAL setting modification**—Modifying the HAL options modifies the system.h file, which is used to compile the BSP library file.
- **HAL initialization**—The HAL is initialized during the execution of the C run-time initialization (crt0) code block. After the crt0 code block runs, the HAL resources are available for your application to use.For more information, refer to the "crt0 Initialization" section.

You can configure the HAL in the BSP Editor. The figure below shows how you specify a UART to be used as the stdio device.

The Tcl script in the example performs the same configuration as in the figure: it specifies a UART to be used as the stdio device.

Figure 4-4: Configuring HAL stdio Device in BSP Editor



Example 4-12: HAL Tcl Configuration Script Example (hal_conf.tcl)

```
#set up stdio file handles to point to a UART
set default_stdio uart1
set_setting hal.stdin $default_stdio
set_setting hal.stdout $default_stdio
set_setting hal.stderr $default_stdio
```

Related Information

- **Nios II Software Build Tools Reference**
- **crt0 Initialization** on page 4-42

Adding Software Packages

Altera supplies several add-on software packages in the Nios II EDS. These software packages are available for your application to use, and can be configured in the BSP Editor from the **Software Packages** tab. The **Software Packages** tab allows you to insert and remove software packages in your BSP, and control software package settings. The software package table at the top of this tab lists each available software package. The table allows you to select the software package version, and enable or disable the software package.

The operating system determines which software packages are available.

The following software packages are provided with the Nios II EDS:

- Host File System—Allows a Nios II system to access a file system that resides on the workstation. For more information, refer to "The Host-Based File System".
- Read-Only Zip File System—Provides access to a simple file system stored in flash memory. For more information, refer to "Read-Only Zip File System".
- NicheStack TCP/IP Stack – Nios II Edition—Enables support of the NicheStack TCP/IP networking stack.

  Note: The stack is provided as is but Altera does not offer additional support.

For more information about the NicheStack TCP/IP networking stack, refer to the Ethernet and the TCP/IP Networking Stack - Nios II Edition chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Ethernet and the NicheStack TCP/IP Stack - Nios II Edition

Using Tcl Scripts with the Nios II BSP Editor

The Nios II BSP Editor supports Tcl scripting. Tcl scripting in the Nios II BSP Editor is a simple but powerful tool that allows you to easily migrate settings from one BSP to another. This feature is especially useful if you have multiple software projects utilizing similar BSP settings. Tcl scripts in the BSP editor allow you to perform the following tasks:

- Regenerate the BSP from the command line
- Export a TCL script from an existing BSP as a starting point for a new BSP
- Recreate the BSP on a different hardware platform
- Examine the Tcl script to improve your understanding of Tcl command usage and BSP settings

You can configure a BSP either by importing your own manually-created Tcl script, or by using a Tcl script exported from the Nios II BSP Editor.

You can apply a Tcl script only at the time that you create the BSP.

Exporting a Tcl Script

To export a Tcl script, follow these steps:

1. Use the Nios II BSP Editor to configure the BSP settings in an existing BSP project.
2. In the Tools menu, click **Export Tcl Script**.
3. Navigate to the directory where you wish to store your Tcl script.
4. Select a file name for the Tcl script.

When creating a Tcl script, the Nios II BSP Editor only exports settings that differ from the BSP defaults. For example, if the only nondefault settings in the BSP are those shown in **Configuring HAL stdio Device in BSP Editor**, the BSP Editor exports the script shown in the example below.

Example 4-13: Tcl Script Exported by BSP Editor

```
#
# This is a generated Tcl script exported
# by a user of the Altera Nios II BSP Editor.
#
# It can be used with the Altera 'nios2-bsp' shell script '--script'
# option to customize a new or existing BSP.
#
#
#
# Exported Setting Changes
#
#
set_setting hal.stdout uart1
set_setting hal.stderr uart1
set_setting hal.stdin uart1
```

For details about default BSP settings, refer to "Specifying BSP Defaults" in the Nios II Software Build Tools chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Nios II Software Build Tools

Importing a Tcl Script to Create a New BSP

The following example illustrates how to configure a new BSP with an imported Tcl script. You import the Tcl script with the Nios II BSP Editor, when you create a new BSP settings file.

In this example, you create the Tcl script by hand, with a text editor. You can also use a Tcl script exported from another BSP, as described in "Exporting a Tcl Script".

To configure a new BSP with a Tcl script, follow these steps:

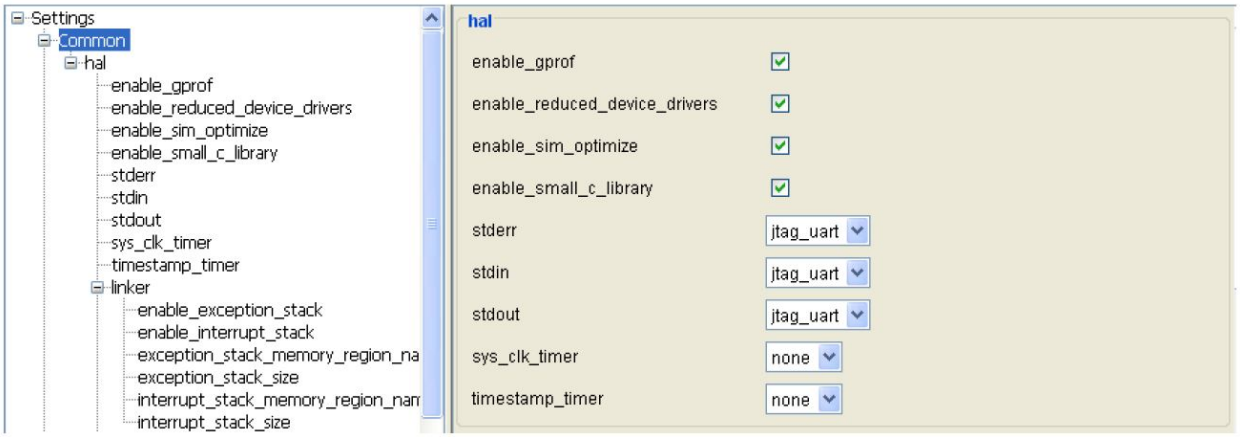
1. With any text editor, create a new file called **example.tcl**.
2. Insert the contents of the example below in the file.
3. In the Nios II BSP Editor, in the File menu, click **New BSP**.
4. In the **BSP Settings File Name** box, select a folder in which to save your new BSP settings file. Accept the default settings file name, **settings.bsp**.
5. In the **Operating System** list, select **Altera HAL**.

6. In the **Additional Tcl script** box, navigate to **example.tcl**.
7. In the **Qsys Information File Name box**, select the **.sopcinfo** file.
8. Click **OK**. The BSP Editor creates the new BSP. The settings modified by **example.tcl** appear as in the figure below.

Example 4-14: Example 2–4. BSP Configuration Tcl Script example.tcl

```
set_setting hal.enable_reduced_device_drivers true
set_setting hal.enable_sim_optimize true
set_setting hal.enable_small_c_library true
set_setting hal.enable_gprof true
```

Figure 4-5: Nios II BSP Settings Configured with example.tcl



Do not attempt to import an Altera HAL Tcl script to a MicroC/OS-II BSP or vice-versa. Doing so could result in unpredictable behavior, such as lost settings. Some BSP settings are OS-specific, making scripts from different OSes incompatible.

For more information about commands that can appear in BSP Tcl scripts, refer to "Software Build Tools Tcl Commands" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Nios II Software Build Tools Reference**
- **Exporting a Tcl Script** on page 4-32

Configuring the Application Project

You configure the application project by specifying source files and a valid BSP project, along with other command-line options to the `nios2-app-generate-makefile` or `nios2-app-update-makefile` commands.

Application Configuration Tips

Use the following tips to increase your efficiency in designing your application project:

1. **Source file inclusion**—To add source files to your project, drag them from a file browser, such as Windows Explorer, and drop them in the **Project Explorer** view in the Nios II Software Build Tools for Eclipse.

   From the command line, several options are available for specifying the source files in your application project. If all your source files are in the same directory, use the `--src-dir` command-line option. If all your source files are contained in a single directory and its subdirectories, use the `--src-rdir` command-line option.
2. **Makefile variables**—When a new project is created in the Nios II Software Build Tools for Eclipse, a makefile is automatically generated in the software project directory. You can modify application makefile variables with the Nios II Application Wizard.

   From the command line, set makefile variables with the `--set <var> <value>` command-line option during configuration of the application project. The variables you can set include the pre- and post-processing settings `BUILD_PRE_PROCESS` and `BUILD_POST_PROCESS` to specify commands to be executed before and after building the application. Examine a generated application makefile to ensure you understand the current and default settings.
3. **Creating top level generation script**—From the command line, simplify the parameterization of your application project by creating a top level shell script to control the configuration. The **create-this-app** scripts in the embedded processor design examples available from the All Design Examples web page are good models for your configuration script.

Related Information

All Design Examples

Linking User Libraries

You can create and use your own user libraries in the Nios II Software Build Tools. The Nios II Software Build Tools for Eclipse includes the Nios II Library wizard, which enables you to create a user library in a GUI environment.

You can also create user libraries in the Nios II Command Shell, as follows:

1. Create the library using the **nios2-lib-generate-makefile** command. This command generates a **public.mk** file.
2. Configure the application project with the new library by running the **nios2-app-generate-makefile** command with the `--use-lib-dir` option. The value for the option specifies the path to the library's **public.mk** file.

Makefiles and the Nios II Software Build Tools for Eclipse

The Nios II Software Build Tools for Eclipse create and manage the makefiles for Nios II software projects. When you create a project, the Nios II Software Build Tools create a makefile based on parameters and settings you select. When you modify parameters and settings, the Nios II Software Build Tools update the makefile to match. BSP makefiles are based on the operating system, BSP settings, selected software packages, and selected drivers.

Nios II BSP makefiles are handled differently from application and user library makefiles. Nios II application and user library makefiles are based on source files that you specify directly. The following changes to an application or user library change the contents of the corresponding makefile:

- Change the application or user library name
- Add or remove source files
- Specify a path to an associated BSP
- Specify a path to an associated user library
- Enable, disable or modify compiler options

For information about BSPs and makefiles, refer to "Makefiles and the Nios II Software Build Tools for Eclipse" in the Getting Started with the Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Getting Started with the Graphical User Interface

Building and Running the Software in Nios II Software Build Tools for Eclipse

Building the Project

After you edit the BSP settings and properties, and generate the BSP (including the makefile), you can build your project. Right-click your project in the **Project Explorer** view and click **Build Project**.

Downloading and Running the Software

To download and run or debug your program, right-click your project in the **Project Explorer** view. To run the program, point to **Run As** and click **Nios II Hardware**.

Before you run your target application, ensure that your FPGA is configured with the target hardware image in your **.sof** file.

Communicating with the Target

The Nios II Software Build Tools for Eclipse provide a console window through which you can communicate with your system. When you use the Nios II Software Build Tools for Eclipse to communicate with the target, characters you input are transmitted to the target line by line. Characters are visible to the target only after you press the Enter key on your keyboard.

If you configured your application to use the `stdio` functions in a UART or JTAG UART interface, you can use the **nios2-terminal** application to communicate with your target subsystem. However, the Nios II Software Build Tools for Eclipse and the **nios2-terminal** application handle input characters very differently.

On the command line, you must use the **nios2-terminal** application to communicate with your target. To start the application, type the following command: **nios2-terminal**

When you use the **nios2-terminal** application, characters you type in the shell are transmitted, one by one, to the target.

Software Debugging in Nios II Software Build Tools for Eclipse

This section describes how to debug a Nios II program using the Nios II Software Build Tools for Eclipse. You can debug a Nios II program on Nios II hardware such as a Nios development board. To debug a software project, right-click the application project name, point to **Debug As** and click **Nios II Hardware**.

Note: Do not select Local C/C++ Application. Nios II projects can only be run and debugged with Nios II run configurations.

For more information about using the Nios II Software Build Tools for Eclipse to debug your application, refer to the Debugging Nios II Designs chapter of the Embedded Design Handbook.

Related Information

Debugging Nios II Designs on page 6-3

Run Time Stack Checking

For debugging purposes, it is useful to enable run-time stack checking, using the hal.enable_runtime_stack_checking BSP setting. When properly used, this setting enables the debugger to take control if the stack collides with the heap or with statically allocated data in memory.

For information about how to use run-time stack checking, refer to “Run-Time Analysis Debug Techniques” and Stack Overflow in the Debugging Nios II Designs chapter. And "Run Time Stack Checking And Exception Debugging" section in the Getting Started with Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*.

For more information about this and other BSP configuration settings, refer to “Settings Managed by the Software Build Tools” in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Getting Started with the Graphical User Interface**
- **Nios II Software Build Tools Reference**
- **Debugging Nios II Designs** on page 6-3

Ensuring Software Project Coherency

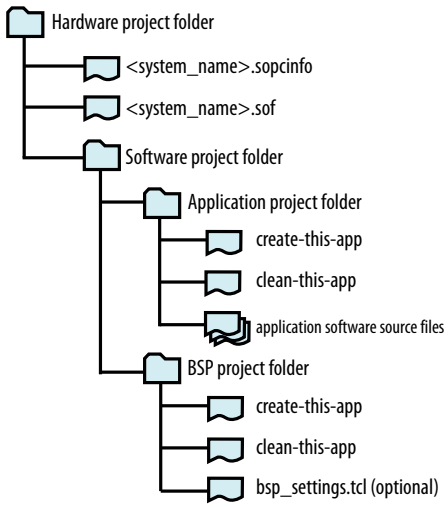
In some engineering environments, maintaining coherency between the software and system hardware projects is difficult. For example, in a mixed team environment in which a hardware engineering team creates new versions of the hardware, independent of the software engineering team, the potential for using the incorrect version of the software on a particular version of the system hardware is high. Such an error may cause engineers to spend time debugging phantom issues. This section discusses several design and software architecture practices that can help you avoid this problem.

Recommended Development Practice

The safest software development practice for avoiding the software coherency problem is to follow a strict hardware and software project hierarchy, and to use scripts to generate your application and BSP projects.

One best practice is to structure your application hierarchy with parallel application project and BSP project folders. In the recommended directory structure below, a top-level hardware project folder includes the Quartus Prime project file, the Qsys-generated files, and the software project folder. The software project folder contains a subfolder for the application project and a subfolder for the BSP project. The application project folder contains a **create-this-app script**, and the BSP project folder contains a **create-this-bsp** script.

Figure 4-6: Recommended Directory Structure



Note: bsp_settings.tcl is a Tcl configuration file. For more information about the Tcl configuration file, refer to "Configuring the BSP Project".

To build your own software project from the command line, create your own **create-this-app** and **create-this-bsp** scripts. Altera recommends that you also create **clean-this-app** and **clean-this-bsp** scripts. These scripts perform the following tasks:

- **create-this-app**—This **bash** script uses the **nios2-app-generate-makefile** command to create the application project, using the application software source files for your project. The script verifies that the BSP project is properly configured (a **settings.bsp** file is present in the BSP project directory), and runs the **create-this-bsp** script if necessary. The Altera-supplied create-this-app scripts that are included in the embedded design examples on the All Design Examples web page of the Altera website provide good models for this script.
- **clean-this-app**—This **bash** script performs all necessary clean-up tasks for the whole project, including the following:
  - Call the application makefile with the clean-all target.
  - Call the **clean-this-bsp** shell script.
- **create-this-bsp**—This **bash** script generates the BSP project. The script uses the **nios2-bsp** command, which can optionally call the configuration script bsp_settings.tcl. The nios2-bsp command references the <*system_name*>**.sopcinfo** file located in the hardware project folder. Running this script creates the BSP project, and builds the BSP library file for the system.
- **clean-this-bsp**—This **bash** script calls the clean target in the BSP project makefile and deletes the **settings.bsp** file.

The complete system generation process, from hardware to BSP and application projects, must be repeated every time a change is made to the system in Qsys. Therefore, defining all your settings in your **create-**

this-bsp script is more efficient than using the Nios II BSP Editor to customize your project. The system generation process follows:

1. Hardware files generation—Using Qsys, write the updated system description to the <*system_name*>**.sopcinfo file**.
2. **Regenerate BSP project**—Generate the BSP project with the **create-this-bsp** script.
3. **Regenerate application project**—Generate the application project with the **create-this-app** script. This script typically runs the **create-this-bsp** script, which builds the BSP project by creating and running the makefile to generate the BSP library file.
4. **Build the system**—Build the system software using the application and BSP makefile scripts. The **create-this-app** script runs make to build both the application project and the BSP library.

To implement this system generation process, Altera recommends that you use the following checklists for handing off responsibility between the hardware and software groups.

Note: This method assumes that the hardware engineering group installs the Nios II EDS. If so, the hardware and software engineering groups must use the same version of the Nios II EDS toolchain.

To hand off the project from the hardware group to the software group, perform the following steps:

1. **Hardware project hand-off**—The hardware group provides copies of the <*system_name*>**.sopcinfo** and <*system_name*>**.sof** files. The software group copies these files to the software group's hardware project folder.
2. **Recreate software project**—The software team recreates the software application for the new hardware by running the **create-this-app** script. This script runs the **create-this-bsp** script.
3. **Build**—The software team runs make in its application project directory to regenerate the software application.

To hand off the project from the software group to the hardware group, perform the following steps:

1. **Clean project directories**—The software group runs the **clean-this-app** script.
2. **Software project folder hand-off**—The software group provides the hardware group with the software project folder structure it generated for the latest hardware version. Ideally, the software project folder contains only the application project files and the application project and BSP generation scripts.
3. **Reconfigure software project**—The hardware group runs the **create-this-app** script to reconfigure the group's application and BSP projects.
4. **Build**—The hardware group runs make in the application project directory to regenerate the software application.

Related Information

- **All Design Examples**
- **Configuring the BSP Project** on page 4-26

Recommended Architecture Practice

Many of the hardware and software coherency issues that arise during the creation of the application software are problems of misplaced peripheral addresses. Because of the flexibility provided by Qsys, almost any peripheral in the system can be assigned an arbitrary address, or have its address modified

during system creation. Implement the following practices to prevent this type of coherency issue during the creation of your software application:

- **Peripheral and Memory Addressing**—The Nios II Software Build Tools automatically generate a system header file, **system.h**, that defines a set of `#define` symbols for every peripheral in the system. These definitions specify the peripheral name, base address location, and address span. If the Memory Management Unit (MMU) is enabled in your Nios II system, verify that the address span for all peripherals is located in direct-mapped memory, outside the memory address range managed by the MMU.

  To protect against coherency issues, access all system peripherals and memory components with their **system.h** name and address span symbols. This method guarantees successful peripheral register access even after a peripheral's addressable location changes.

  For example, if your system includes a UART peripheral named UART1, located at address 0x1000, access the UART1 registers using the **system.h** address symbol `(iowr_32(UART1_BASE, 0x0, 0x10101010))` rather than using its address `(iowr_32(0x1000, 0x0, 0x10101010))`.
- **Checking peripheral values with the preprocessor**—If you work in a large team environment, and your software has a dependency on a particular hardware address, you can create a set of C preprocessor `#ifdef` statements that validate the hardware during the software compilation process. These #ifdef statements validate the `#define` values in the system.h file for each peripheral.

  For example, for the peripheral UART1, assume the #define values in **system.h** appear as follows:

  ```
  #define UART1_NAME "/dev/uart1"
  #define UART1_BASE 0x1000
  #define UART1_SPAN 32
  #define UART1_IRQ 6
  . . .
  ```

  In your C/C++ source files, add a preprocessor macro to verify that your expected peripheral settings remain unchanged in the hardware configuration. For example, the following code checks that the base address of UART1 remains at the expected value:

  ```
  #if (UART1_BASE != 0x1000)
  #error UART should be at 0x1000, but it is not
  #endif
  ```
- **Ensuring coherency with the System ID core**—Use the System ID core. The System ID core is an Qsys peripheral that provides a unique identifier for a generated hardware system. This identifier is stored in a hardware register readable by the Nios II processor. This unique identifier is also stored in the **.sopcinfo** file, which is then used to generate the BSP project for the system. You can use the system ID core to ensure coherency between the hardware and software by either of the following methods:
  - The first method is optionally implemented during system software development, when the Executable and Linking Format (**.elf**) file is downloaded to the Nios II target. During the software download process, the value of the system ID core is checked against the value present in the BSP library file. If the two values do not match, this condition is reported. If you know that the system ID difference is not relevant, the system ID check can be overridden to force a download. Use this override with extreme caution, because a mismatch between hardware and software can lead you to waste time trying to resolve nonexistent bugs.
  - The second method for using the system ID peripheral is useful in systems that do not have a Nios II debug port, or in situations in which running the Nios II software download utilities is not practical. In this method you use the C function `alt_avalon_sysid_test()`. This function reports whether the hardware and software system IDs match.

  For more information about the System ID core, refer to the System ID Core chapter of the *Embedded Peripherals IP User Guide*.

Related Information

Embedded Peripherals IP User Guide

Developing With the Hardware Abstraction Layer

The HAL for the Nios II processor is a lightweight run-time environment that provides a simple device driver interface for programs to communicate with the underlying hardware. The HAL API is integrated with the ANSI C standard library. The HAL API allows you to access devices and files using familiar C library functions.

Overview of the HAL

This section describes how to use HAL services in your Nios II software. It provides information about the HAL configuration options, and the details of system startup and HAL services in HAL-based applications.

HAL Configuration Options

To support the Nios II software development flow, the HAL BSP library is self-configuring to some extent. By design, the HAL attempts to enable as many services as possible, based on the peripherals present in the system hardware. This approach provides your application with the least restrictive environment possible—a useful feature during the product development and board bringup cycle.

The HAL is configured with a group of settings whose values are determined by Tcl commands, which are called during the creation of the BSP project.

As mentioned in "Configuring the BSP Project", Altera recommends you create a separate Tcl file that contains your HAL configuration settings.

HAL configuration settings control the boot loading process, and provide detailed control over the initialization process, system optimization, and the configuration of peripherals and services. For each of these topics, this section provides pointers to the relevant material elsewhere in this section.

Related Information

Configuring the BSP Project on page 4-26

Configuring the Boot Environment

Your particular system may require a boot loader to configure the application image before it can begin execution. For example, if your application image is stored in flash memory and must be copied to volatile memory for execution, a boot loader must configure the application image in the volatile memory. This configuration process occurs before the HAL BSP library configuration routines execute, and before the crt0 code block executes. A boot loader implements this process.

For more information, refer to "Linking Applications" and "Application Boot Loading and Programming System Memory".

Related Information

- **Linking Applications** on page 4-59
- **Application Boot Loading and Programming System Memory** on page 5-3

Controlling HAL Initialization

As noted in "HAL Initialization", although most application debugging begins in the `main()` function, some tasks, such as debugging device driver initialization, require the ability to control overall system initialization after the crt0 initialization routine runs and before `main()` is called.

For an example of this kind of application, refer to the hello_alt_main software example design supplied with the Nios II EDS installation.

Related Information

HAL Initialization on page 4-43

Minimizing the Code Footprint and Increasing Performance

For information about increasing your application's performance, or minimizing the code footprint, refer to "Software Application Optimization".

Related Information

Software Application Optimization on page 7-16

Configuring Peripherals and Services

For information about configuring and using HAL services, refer to "HAL Peripheral Services".

Related Information

HAL Peripheral Services on page 4-44

System Startup in HAL-Based Applications

System startup in HAL-based applications is a three-stage process. First, the system initializes, then the `crt0` code section runs, and finally the HAL services initialize. The following sections describe these three system-startup stages.

System Initialization

The system initialization sequence begins when the system powers up. The initialization sequence steps for FPGA designs that contain a Nios II processor are the following:

1. **Hardware reset event**—The board receives a power-on reset signal, which resets the FPGA.
2. **FPGA configuration**—The FPGA is programmed with a .sof file, from a specialized configuration memory or an external hardware master. The external hardware master can be a CPLD device or an external processor.
3. **System reset**—The Qsys system, composed of one or more Nios II processors and other peripherals, receives a hardware reset signal and enters the components' combined reset state.
4. **Nios II processor(s)**—Each Nios II processor jumps to its preconfigured reset address, and begins running instructions found at this address.
5. **Boot loader or program code**—Depending on your system design, the reset address vector contains a packaged boot loader, called a boot image, or your application image. Use the boot loader if the application image must be copied from non-volatile memory to volatile memory for program execution. This case occurs, for example, if the program is stored in flash memory but runs from SDRAM. If no boot loader is present, the reset vector jumps directly to the .crt0 section of the application image. Do not use a boot loader if you wish your program to run in-place from non-volatile or preprogrammed memory.

   For additional information about both of these cases, refer to "Application Boot Loading and Programming System Memory".
6. **crt0 execution**—After the boot loader executes, the processor jumps to the beginning of the program's initialization block—the `.crt0` code section. The function of the `crt0` code block is detailed in the next section.

Related Information

Application Boot Loading and Programming System Memory on page 5-3

crt0 Initialization

The `crt0` code block contains the C run-time initialization code—software instructions needed to enable execution of C or C++ applications. The `crt0` code block can potentially be used by user-defined assembly language procedures as well. The Altera-provided crt0 block performs the following initialization steps:

1. **Calls alt_load macros**—If the application is designed to run from flash memory (the .text section runs from flash memory), the remaining sections are copied to volatile memory.

   For additional information, refer to "Configuring the Boot Environment".
2. **Initializes instruction cache**—If the processor has an instruction cache, this cache is initialized. All instruction cache lines are zeroed (without flushing) with the initi instruction.

   Note: Qsys determines the processors that have instruction caches, and configures these caches at system generation. The Nios II Software Build Tools insert the instruction-cache initialization code block if necessary.
3. **Initializes data cache**—If the processor has a data cache, this cache is initialized. All data cache lines are zeroed (without flushing) with the `initd` instruction. As for the instruction caches, this code is enabled if the processor has a data cache.
4. **Sets the stack pointer**—The stack pointer is initialized. You can set the stack pointer address.

   For additional information refer to "HAL Linking Behavior".
5. **Clears the .bss section**—The .bss section is initialized to all zeroes. You can set the .bss section address.

   For additional information refer to "HAL Linking Behavior".
6. **Initializes stack overflow protection**—Stack overflow checking is initialized.

   For additional information, refer to "Software Debugging in Nios II Software Build Tools for Eclipse".
7. **Jumps to** `alt_main()`—The processor jumps to the `alt_main()` function, which begins initializing the HAL BSP run-time library.

   Note: If you use a third-party RTOS or environment for your BSP library file, the `alt_main()` function could be different than the one provided by the Nios II EDS.

If you use a third-party compiler or library, the C run-time initialization behavior may differ from this description.

The `crt0` code includes initialization short-cuts only if you perform hardware simulations of your design. You can control these optimizations by turning `hal.enable_sim_optimize` on or off.

For information about the `hal.enable_sim_optimize` BSP setting, refer to "Settings Managed by the Software Build Tools" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

The **crt0.S** source file is located in the <*Altera tools installation*>**/ip/altera/nios2_ip/altera_nios2/HAL/src** directory.

Related Information

- **Nios II Software Build Tools Reference**
- **Configuring the Boot Environment** on page 4-40
- **HAL Linking Behavior** on page 4-59
- **Software Debugging in Nios II Software Build Tools for Eclipse** on page 4-35

HAL Initialization

As for any other C program, the first part of the HAL's initialization is implemented by the Nios II processor's `crt0.S` routine. For more information, see "crt0 Initialization". After `crt0.S` completes the C run-time initialization, it calls the HAL `alt_main()` function, which initializes the HAL BSP run-time library and subsystems.

The HAL `alt_main()` function performs the following steps:

1. **Initializes interrupts**—Sets up interrupt support for the Nios II processor (with the `alt_irq_init()` function).
2. **Starts MicroC/OS-II**—Starts the MicroC/OS-II RTOS, if this RTOS is configured to run (with the `ALT_OS_INIT` and `ALT_SEM_CREATE` functions). For additional information about MicroC/OS-II use and initialization, refer to "Selecting the Operating System (HAL versus MicroC/OS-II RTOS)".
3. **Initializes device drivers**—Initializes device drivers (with the `alt_sys_init()` function). The Nios II Software Build Tools automatically find all peripherals supported by the HAL, and automatically insert a call to a device configuration function for each peripheral in the `alt_sys_init()` code. To override this behavior, you can disable a device driver with the Nios II BSP Editor, in the Drivers tab.

   For information about enabling and disabling device drivers, refer to "Using the BSP Editor" in the Getting Started with the Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*.

   To disable a driver from the Nios II Command Shell, use the following option to the **nios2-bsp** script:

   ```
   --cmd set_driver <peripheral_name> none
   ```

   For information about removing a device configuration function, and other methods of reducing the BSP library size, refer to **Table 4-1**.
4. **Configures stdio functions**—Initializes stdio services for `stdin`, `stderr`, and `stdout`. These services enable the application to use the GNU newlib stdio functions and maps the file pointers to supported character devices.For more information about configuring the stdio services, refer to "Character Mode Devices".
5. `Initializes C++ CTORS and DTORS`—Handles initialization of C++ constructor and destructor functions. These function calls are necessary if your application is written in the C++ programming language. By default, the HAL configuration mechanism enables support for the C++ programming language. Disabling this feature reduces your application's code footprint, as noted in "Software Application Optimization".

   The Nios II C++ language support depends on the GCC tool chain. The Nios II GCC 4 C++ tool chain supports polymorphism, friendship and inheritance, multiple inheritance, virtual base classes, run-time type information (typeid), the mutable type qualifier, namespaces, templates, new-and-delete style dynamic memory allocation, operator overloading, and the Standard Template Library (STL). Exceptions and new-style dynamic casts are not supported.
6. **Calls main()**—Calls function `main()`, or application program. Most applications are constructed using a `main()` function declaration, and begin execution at this function.

   If you use a BSP that is not based on the HAL and need to initialize it after the `crt0.S` routine runs, define your own `alt_main()` function. For an example, see the `main()` and `alt_main()` functions in the **hello_alt_main.c** file at *<Nios II EDS install dir>***examples\software\hello_alt_main**.

   After you generate your BSP project, the **alt_main.c** source file is located in the **HAL/src** directory.

Related Information

- **Getting Started with the Graphical User Interface**
- **crt0 Initialization** on page 4-42

- **Selecting the Operating System (HAL versus MicroC/OS-II RTOS)** on page 4-26
- **Character Mode Devices** on page 4-46
- **Software Application Optimization** on page 7-16

HAL Peripheral Services

The HAL provides your application with a set of services, typically relying on the presence of a hardware peripheral to support the services. By default, if you configure your HAL BSP project from the command-line by running the **nios2-bsp** script, each peripheral in the system is initialized, operational, and usable as a service at the entry point of your C/C++ application (`main()`).

This section describes the core set of Altera-supplied, HAL-accessible peripherals and the services they provide for your application. It also describes application design guidelines for using the supplied service, and background and configuration information, where appropriate.

For more information about the HAL peripheral services, refer to the Developing Programs Using the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook*. For more information about HAL BSP configuration settings, refer to the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Nios II Software Build Tools Reference**
- **Developing Programs Using the Hardware Abstraction Layer**

Timers

The HAL provides two types of timer services, a system clock timer and a timestamp timer. The system clock timer is used to control, monitor, and schedule system events. The timestamp variant is used to make high performance timing measurements. Each of these timer services is assigned to a single Altera Avalon Timer peripheral.

For more information about this peripheral, refer to the Interval Timer Core chapter of the *Embedded Peripherals IP User Guide*.

Related Information

Embedded Peripherals IP User Guide

System Clock Timer

The system clock timer resource is used to trigger periodic events (alarms), and as a timekeeping device that counts system clock ticks. The system clock timer service requires that a timer peripheral be present in the Qsys system. This timer peripheral must be dedicated to the HAL system clock timer service.

Note: Only one system clock timer service may be identified in the BSP library. This timer should be accessed only by HAL supplied routines.

The `hal.sys_clk_timer` setting controls the BSP project configuration for the system clock timer. This setting configures one of the timers available in your Qsys design as the system clock timer.

Altera provides separate APIs for application-level system clock functionality and for generating alarms.

Application-level system clock functionality is provided by two separate classes of APIs, one Nios II specific and the other Unix-like. The Altera function `alt_nticks` returns the number of clock ticks that have elapsed. You can convert this value to seconds by dividing by the value returned by the `alt_ticks_per_second()` function. For most embedded applications, this function is sufficient for rudimentary time keeping.

The POSIX-like `gettimeofday()` function behaves differently in the HAL than on a Unix workstation. On a workstation, with a battery backed-up, real-time clock, this function returns an absolute time value, with the value zero representing 00:00 Coordinated Universal Time (UTC), January 1, 1970, whereas in the HAL, this function returns a time value starting from system power-up. By default, the function assumes system power-up to have occurred on January 1, 1970. Use the `settimeofday()` function to correct the HAL gettimeofday() response. The `times()` function exhibits the same behavior difference.

Consider the following common issues and important points before you implement a system clock timer:

- **System Clock Resolution**—The timer's period value specifies the rate at which the HAL BSP project increments the internal variable for the system clock counter. If the system clock increments too slowly for your application, you can decrease the timer's period in Qsys.
- **Rollover**—The internal, global variable that stores the number of system clock counts (since reset) is a 32-bit unsigned integer. No rollover protection is offered for this variable. Therefore, you should calculate when the rollover event will occur in your system, and plan the application accordingly.
- **Performance Impact**—Every clock tick causes the execution of an interrupt service routine. Executing this routine leads to a minor performance penalty. If your system hardware specifies a short timer period, the cumulative interrupt latency may impact your overall system performance.

The alarm API allows you to schedule events based on the system clock timer, in the same way an alarm clock operates. The API consists of the `alt_alarm_start()` function, which registers an alarm, and the `alt_alarm_stop()` function, which disables a registered alarm.

Consider the following common issues and important points before you implement an alarm:

- **Interrupt Service Routine (ISR) context**—A common mistake is to program the alarm callback function to call a service that depends on interrupts being enabled (such as the `printf()` function). This mistake causes the system to deadlock, because the alarm callback function occurs in an interrupt context, while interrupts are disabled.
- **Resetting the alarm**—The callback function can reset the alarm by returning a nonzero value. Internally, the `alt_alarm_start()` function is called by the callback function with this value.
- **Chaining**—The `alt_alarm_start()` function is capable of handling one or more registered events, each with its own callback function and number of system clock ticks to the alarm.
- **Rollover**—The alarm API handles clock rollover conditions for registered alarms seamlessly.

A good timer period for most embedded systems is 50 ms. This value provides enough resolution for most system events, but does not seriously impact performance nor roll over the system clock counter too quickly.

Timestamp Timer

The timestamp timer service provides applications with an accurate way to measure the duration of an event in the system. The timestamp timer service requires that a timer peripheral be present in the Qsys system. This timer peripheral must be dedicated to the HAL timestamp timer service.

Only one timestamp timer service may be identified in the BSP library file. This timer should be accessed only by HAL supplied routines.

The `hal.timestamp_timer` setting controls the BSP configuration for the timer. This setting configures one of the timers available in the Qsys design as the timestamp timer.

Altera provides a timestamp API. The timestamp API is very simple. It includes the `alt_timestamp_start()` function, which makes the timer operational, and the `alt_timestamp()` function, which returns the current timer count.

Consider the following common issues and important points before you implement a timestamp timer:

- **Timer Frequency**—The timestamp timer decrements at the clock rate of the clock that feeds it in the Qsys system. You can modify this frequency in Qsys.
- **Rollover**—The timestamp timer has no rollover event. When the alt_timestamp() function returns the value 0, the timer has run down.
- **Maximum Time**—The timer peripheral has 32 bits available to store the timer value. Therefore, the maximum duration a timestamp timer can count is $((1/\textit{timer frequency}) \times 2^{32})$ seconds.

For more information about the APIs that control the timestamp and system clock timer services, refer to the HAL API Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

HAL API Reference

Character Mode Devices

stdin, stdout, and stderr

The HAL can support the stdio functions provided in the GNU newlib library. Using the stdio library allows you to communicate with your application using functions such as `printf()` and `scanf()`.

Currently, Altera supplies two system components that can support the `stdio` library, the UART and JTAG UART components. These devices can function as standard I/O devices.

To enable this functionality, use the `--default_stdio` <*device*> option during Nios II BSP configuration. The `stdin` character input file variable and the `stdout` and `stderr` character output file variables can also be individually configured with the HAL BSP settings `hal.stdin`, `hal.stdout`, and `hal.stderr`.

Make sure that you assign values individually for each of the `stdin`, `stdout`, and `stderr` file variables that you use.

After your target system is configured to use the `stdin`, `stdout`, and `stderr` file variables with either the UART or JTAG UART peripheral, you can communicate with the target Nios II system with the Nios II EDS development tools. For more information about performing this task, refer to "Communicating with the Target".

For more information about the `--default_stdio` <*device*> option, refer to "Nios II Software Build Tools Utilities" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Nios II Software Build Tools Reference**
- **Communicating with the Target** on page 4-35

Blocking versus Non-Blocking I/O

Character mode devices can be configured to operate in blocking mode or non-blocking mode. The mode is specified in the device's file descriptor. In blocking mode, a function call to read from the device waits until the device receives new data. In non-blocking mode, the function call to read new data returns immediately and reports whether new data was received. Depending on the function you use to read the file handle, an error code is returned, specifying whether or not new data arrived.

The UART and JTAG UART components are initialized in blocking mode. However, each component can be made non-blocking with the `fnctl` or the `ioctl()` function, as seen in the following open system call, which specifies that the device being opened is to function in non-blocking mode:

```
fd = open ("/dev/<your uart name>", O_NONBLOCK | O_RDWR);
```

The `fnctl()` system call shown in the example below specifies that a device that is already open is to function in non-blocking mode:

Example 4-15: fnctl() System Call

```
/* You can specify <file_descriptor> to be
* STDIN_FILENO, STDOUT_FILENO, or STDERR_FILENO
* if you are using STDIO
*/
fnctl(<file_descriptor>, F_SETFL, O_NONBLOCK);
```

Example 4-16: Non-Blocking Device Code Fragment

```
input_chars[128];
return_chars = scanf("%128s", &input_chars);
if(return_chars == 0)
{
if(errno != EWOULDBLOCK)
{
/* check other errnos */
}
}
else
{
/* process received characters */
}
```

The behavior of the UART and JTAG UART peripherals can also be modified with an `ioctl()` function call. The `ioctl()` function supports the following parameters:

- For UART peripherals:
  - TIOCMGET (reports baud rate of UART)
  - TIOCMSET (sets baud rate of UART)
- For JTAG UART peripherals:
  - TIOCSTIMEOUT (timeout value for connecting to workstation)
  - TIOCGCONNECTED (find out whether host is connected)

The `altera_avalon_uart_driver.enable_ioctl` BSP setting enables and disables the `ioctl()` function for the UART peripherals. The `ioctl()` function is automatically enabled for the JTAG UART peripherals.

The `ioctl()` function is not compatible with the `altera_avalon_uart_driver.enable_small_driver` and `hal.enable_reduced_driver` BSP settings. If either of these settings is enabled, `ioctl()` is not implemented.

Adding Your Own Character Mode Device

If you have a custom device capable of character mode operation, you can create a custom device driver that the `stdio` library functions can use.

For information about how to develop the device driver, refer to *AN459: Guidelines for Developing a Nios II HAL Device Driver*.

Related Information

AN459: Guidelines for Developing a Nios II HAL Device Driver

Flash Memory Devices

The HAL BSP library supports parallel common flash interface (CFI) memory devices and Altera erasable, programmable, configurable serial (EPCS) flash memory devices. A uniform API is available for both flash memory types, providing read, write, and erase capabilities.

Memory Initialization, Querying, and Device Support

Every flash memory device is queried by the HAL during system initialization to determine the kind of flash memory and the functions that should be used to manage it. This process is automatically performed by the `alt_sys_init()` function, if the device drivers are not explicitly omitted and the small driver configuration is not set.

After initialization, you can query the flash memory for status information with the `alt_flash_get_flash_info()` function. This function returns a pointer to an array of flash region structures—C structures of type `struct flash_region`—and the number of regions on the flash device.

For additional information about the `struct flash_region` structure, refer to the source file **HAL/inc/sys/alt_flash_types.h** in the BSP project directory.

Accessing the Flash Memory

The `alt_flash_open()` function opens a flash memory device and returns a descriptor for that flash memory device. After you complete reading and writing the flash memory, call the `alt_flash_close()` function to close it safely.

The HAL flash memory device model provides you with two flash access APIs, one simple and one fine-grained The simple API takes a buffer of data and writes it to the flash memory device, erasing the sectors if necessary. The fine-grained API enables you to manage your flash device on a block-by-block basis.

Both APIs can be used in the system. The type of data you store determines the most useful API for your application. The following general design guidelines help you determine which API to use for your data storage needs:

Simple API—This API is useful for storing arbitrary streams of bytes, if the exact flash sector location is not important. Examples of this type of data are log or data files generated by the system during run-time, which must be accessed later in a continuous stream somewhere in flash memory.

Fine-Grained API—This API is useful for storing units of data, or data sets, which must be aligned on absolute sector boundaries. Examples of this type of data include persistent user configuration values, FPGA hardware images, and application images, which must be stored and accessed in a given flash sector (or sectors).

For examples that demonstrate the use of APIs, refer to the "Using Flash Devices" section in the Developing Programs Using the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Developing Programs Using the Hardware Abstraction Layer

Configuration and Use Limitations

If you use flash memories in your system, be aware of the following properties of this memory:

- **Code Storage**—If your application runs code directly from the flash memory, the flash manipulation functions are disabled. This setting prevents the processor from erasing the memory that holds the code it is running. In this case, the symbols `ALT_TEXT_DEVICE`, `ALT_RODATA_DEVICE`, and `ALT_EXCEPTIONS_DEVICE` must all have values different from the flash memory peripheral. (Note that each of these #define symbols names a memory device, not an address within a memory device).
- **Small Driver**—If the small driver flag is set for the software—the `hal.enable_reduced_device_drivers` setting is enabled—then the flash memory peripherals are not automatically initialized. In this case, your application must call the initialization routines explicitly.
- **Thread safety**—Most of the flash access routines are not thread-safe. If you use any of these routines, construct your application so that only one thread in the system accesses these function.
- **EPCS flash memory limitations**—The Altera EPCS memory has a serial interface. Therefore, it cannot run Nios II instructions and is not visible to the Nios II processor as a standard random-access memory device. Use the Altera-supplied flash memory access routines to read data from this device.
- **File System**—The HAL flash memory API does not support a flash file system in which data can be stored and retrieved using a conventional file handle. However, you can store your data in flash memory before you run your application, using the read-only zip file system and the Nios II flash programmer utility. For information about the read-only zip file system, refer to "Read-Only Zip File System".

For more information about the configuration and use limitations of flash memory, refer to the "Using Flash Devices" section in the Developing Programs Using the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook*. For more information about the API for the flash memory access routines, refer to the HAL API Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **HAL API Reference**
- **Developing Programs Using the Hardware Abstraction Layer**
- **Read-Only Zip File System** on page 4-53

Direct Memory Access Devices

The HAL Direct Memory Access (DMA) model uses DMA transmit and receive channels. A DMA operation places a transaction request on a channel. A DMA peripheral can have a transmit channel, a receive channel, or both. This section describes three possible hardware configurations for a DMA peripheral, and shows how to activate each kind of DMA channel using the HAL memory access functions.

The DMA peripherals are initialized by the `alt_sys_init()` function call, and are automatically enabled by the **nios2-bsp** script.

DMA Configuration and Use Model

The following examples illustrate use of the DMA transmit and receive channels in a system. The information complements the information available in "Using DMA Devices" in the Developing Programs Using the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook*.

Regardless of the DMA peripheral connections in the system, initialize a transmit channel by running the `alt_dma_txchan_open()` function, and initialize a receive DMA channel by running the `alt_dma_rxchan_open()` function. The following sections describe the use model for some specific cases.

Related Information

Developing Programs Using the Hardware Abstraction Layer

RX-Only DMA Component

A typical RX-only DMA component moves the data it receives from another component to memory. In this case, the receive channel of the DMA peripheral reads continuously from a fixed location in memory, which is the other peripheral's data register. The following sequence of operations directs the DMA peripheral:

1. Open the DMA peripheral—Call the `alt_dma_rxchan_open()` function to open the receive DMA channel.
2. Enable DMA `ioctl` operations—Call the `alt_dma_rxchan_ioctl()` function to set the `ALT_DMA_RX_ONLY_ON` flag. Use the `ALT_DMA_SET_MODE_<n>` flag to set the data width to match that of the other peripheral's data register.
3. Configure the other peripheral to run—The Nios II processor configures the other peripheral to begin loading new data in its data register.
4. Queue the DMA transaction requests—Call the `alt_avalon_dma_prepare()` function to begin a DMA operation. In the function call, you specify the DMA receive channel, the other peripheral's data register address, the number of bytes to transfer, and a callback function to run when the transaction is complete.

TX-Only DMA Component

A typical TX-only DMA component moves data from memory to another component. In this case, the transmit channel of the DMA peripheral writes continuously to a fixed location in memory, which is the other peripheral's data register. The following sequence of operations directs the DMA peripheral:

1. Open the DMA peripheral—Call the `alt_dma_txchan_open()` function to open the transmit DMA channel.
2. Enable DMA `ioctl` operations—Call the `alt_dma_txchan_ioctl()` function to set the `ALT_DMA_TX_ONLY_ON` flag. Use the `ALT_DMA_SET_MODE_<n>` flag to set the data width to match that of the other peripheral's data register.
3. Configure the other peripheral to run—The Nios II processor configures the other peripheral to begin receiving new data in its data register.
4. Queue the DMA transaction requests—Call the `alt_avalon_dma_send()` function to begin a DMA operation. In the function call, you specify the DMA transmit channel, the other peripheral's data register address, the number of bytes to transfer, and a callback function to run when the transaction is complete.

RX and TX DMA Component

A typical RX and TX DMA component performs memory-to-memory copy operations. The application must open, configure, and assign transaction requests to both DMA channels explicitly. The following sequence of operations directs the DMA peripheral:

1. Open the DMA RX channel—Call the `alt_dma_rxchan_open()` function to open the DMA receive channel.
2. Enable DMA RX `ioctl` operations—Call the `alt_dma_rxchan_ioctl()` function to set the `ALT_DMA_RX_ONLY_OFF` flag. Use the `ALT_DMA_SET_MODE_<n>` flag to set the data width to the correct value for the memory transfers.
3. Open the DMA TX channel—Call the `alt_dma_txchan_open()` function to open the DMA transmit channel.
4. Enable DMA TX `ioctl` operations—Call the `alt_dma_txchan_ioctl()` function to set the `ALT_DMA_TX_ONLY_OFF` flag. Use the `ALT_DMA_SET_MODE_<n>` flag to set the data width to the correct value for the memory transfers.
5. Queue the DMA RX transaction requests—Call the `alt_avalon_dma_prepare()` function to begin a DMA RX operation. In the function call, you specify the DMA receive channel, the address from which to begin reading, the number of bytes to transfer, and a callback function to run when the transaction is complete.
6. Queue the DMA TX transaction requests—Call the `alt_avalon_dma_send()` function to begin a DMA TX operation. In the function call, you specify the DMA transmit channel, the address to which to begin writing, the number of bytes to transfer, and a callback function to run when the transaction is complete.

The DMA peripheral does not begin the transaction until the DMA TX transaction request is issued.

For examples of DMA device use, refer to "Using DMA Devices" in the Developing Programs Using the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Developing Programs Using the Hardware Abstraction Layer

DMA Data-Width Parameter

The DMA data-width parameter is configured in Qsys to specify the widths that are supported. In writing the software application, you must specify the width to use for a particular transaction. The width of the data you transfer must match the hardware capability of the component.

Consider the following points about the data-width parameter before you implement a DMA peripheral:

- Peripheral width—When a DMA component moves data from another peripheral, the DMA component must use a single-operation transfer size equal to the width of the peripheral's data register.
- Transfer length—The byte transfer length specified to the DMA peripheral must be a multiple of the data width specified.
- Odd transfer sizes—If you must transfer an uneven number of bytes between memory and a peripheral using a DMA component, you must divide up your data transfer operation. Implement the longest allowed transfer using the DMA component, and transfer the remaining bytes using the Nios II processor. For example, if you must transfer 1023 bytes of data from memory to a peripheral with a 32-bit data register, perform 255 32-bit transfers with the DMA and then have the Nios II processor write the remaining 3 bytes.

Configuration and Use Limitations

If you use DMA components in your system, be aware of the following properties of these components:

- **Hardware configuration**—The following aspects of the hardware configuration of the DMA peripheral determine the HAL service:
  - DMA components connected to peripherals other than memory support only half of the HAL API (receive or transmit functionality). The application software should not attempt to call API functions that are not available.
  - The hardware parameterization of the DMA component determines the data width of its transfers, a value which the application software must take into account.
- **IOCTL control**—The DMA `ioctl()` function call enables the setting of a single flag only. To set multiple flags for a DMA channel, you must call `ioctl()` multiple times.
- **DMA transaction slots**—The current driver is limited to four transaction slots. If you must increase the number of transaction slots, you can specify the number of slots using the macro `ALT_AVALON_DMA_NSLOTS`. The value of this macro must be a power of two.
- `Interrupts`—The HAL DMA service requires that the DMA peripheral's interrupt line be connected in the system.
- **User controlled DMA accesses**—If the default HAL DMA access routines are too unwieldy for your application, you can create your own access functions. For information about how to remove the default HAL DMA driver routines, refer to "Reducing Code Size".

For more information about the HAL API for accessing DMA devices, refer to "Using DMA Devices" in the Developing Programs Using the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook* and to the HAL API Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Developing Programs Using the Hardware Abstraction Layer**
- **Reducing Code Size** on page 7-19

Files and File Systems

The HAL provides two simple file systems and an API for dealing with file data. The HAL uses the GNU newlib library's file access routines, found in file.h, to provide access to files. In addition, the HAL provides the following file systems:

- Host-based file system—Enables a Nios II system to access the host workstation's file system
- Read-only zip file system—Enables simple access to preconfigured data in the Nios II system memory

Several more conventional file systems that support both read and write operations are available through third-party vendors. For up-to-date information about the file system solutions available for the Nios II processor, visit the Nios II Processor page of the Altera website, and look for **Altera Embedded Alliance**.

To make either of these software packages visible to your application, you must enable it in the BSP. You can enable a software package either in the BSP Editor, or from the command line. The names that specify the host-based file system and read-only zip file system packages are `altera_hostfs` and `altera_ro_zipfs`, respectively.

Related Information

Nios II Processor

The Host-Based File System

The host-based file system enables the Nios II system to manipulate files on a workstation through a JTAG connection. The API is a transparent way to access data files. The system does not require a physical block device.

Consider the following points about the host-based file system before you use it:

- **Communication speed**—Reading and writing large files to the Nios II system using this file system is slow.
- **Debug use mode**—The host-based file system is only available during debug sessions from the Nios II debug perspective. Therefore, you should use the host-based file system only during system debugging and prototyping operations.
- **Incompatibility with direct drivers**—The host-based file system only works if the HAL BSP library is configured with direct driver mode disabled. However, enabling this mode reduces the size of the application image. For more information, refer to "Software Application Optimization".

For more information about the host file system, refer to "Using File Subsystems" in the Developing Programs Using the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Software Application Optimization** on page 7-16
- **Developing Programs Using the Hardware Abstraction Layer**

Read-Only Zip File System

The read-only zip file system is a lightweight file system for the Nios II processor, targeting flash memory.

Consider the following points about the read-only zip file system before you use it:

- **Read-Only**—The read-only zip file system does not implement writes to the file system.
- **Configuring the file system**—To create the read-only zip file system you must create a binary file on your workstation and use the Nios II flash programmer utility to program it in the Nios II system.
- **Incompatibility with direct drivers**—The read-only zip file system only works if the HAL BSP library is configured with direct driver mode disabled. However, enabling this mode reduces the size of the application image. For more information, refer to "Software Application Optimization".

For more information, refer to the Read-Only Zip File System and Developing Programs Using the Hardware Abstraction Layer chapters of the *Nios II Gen2 Software Developer's Handbook*. Also the read-only zip file system Nios II software example design listed in "Nios II Design Example Scripts" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Nios II Software Build Tools Reference**
- **Software Application Optimization** on page 7-16
- **Developing Programs Using the Hardware Abstraction Layer**
- **Read-Only Zip File System**

Unsupported Devices

The HAL provides a wide variety of native device support for Altera-supplied peripherals. However, your system may require a device or peripheral that Altera does not provide. In this case, one or both of the following two options may be available to you:

- Obtain a device through Altera's third-party program
- Incorporate your own device

Altera's third-party program information is available on the Nios II embedded software partners page. Refer to the Nios II Processor page of the Altera website, and look for **Altera Embedded Alliance**.

Incorporating your own custom peripheral is a two-stage process. First you must incorporate the peripheral in the hardware, and then you must develop a device driver.

For more information about how to incorporate a new peripheral in the hardware, refer to the *Nios II Hardware Development Tutorial*. For more information about how to develop a device driver, refer to the Developing Device Drivers for the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook* and *AN459: Guidelines for Developing a Nios II HAL Device Driver*.

Related Information

- **AN459: Guidelines for Developing a Nios II HAL Device Driver**
- **Nios II Processor**
- **Developing Device Drivers for the Hardware Abstraction Layer**
- **Nios II Hardware Development Tutorial**

Accessing Memory With the Nios II Processor

It can be difficult to create software applications that program the Nios II processor to interact correctly with data and instruction caches when it reads and writes to peripherals and memories. There are also subtle differences in how the different Nios II processor cores handle these operations, that can cause problems when you migrate from one Nios II processor core to another.

This section helps you avoid the most common pitfalls. It provides background critical to understanding how the Nios II processor reads and writes peripherals and memories, and describes the set of software utilities available to you, as well as providing sets of instructions to help you avoid some of the more common problems in programming these read and write operations.

Creating General C/C++ Applications

You can write most C/C++ applications without worrying about whether the processor's read and write operations bypass the data cache. However, you do need to make sure the operations do not bypass the data cache in the following cases:

- Your application must guarantee that a read or write transaction actually reaches a peripheral or memory. This guarantee is critical for the correct functioning of a device driver interrupt service routine, for example.
- Your application shares a block of memory with another processor or Avalon interface master peripheral.

Accessing Peripherals

If your application accesses peripheral registers, or performs only a small set of memory accesses, Altera recommends that you use the default HAL I/O macros, IORD and IOWR. These macros guarantee that the accesses bypass the data cache.

Two types of cache-bypass macros are available. The HAL access routines whose names end in _32DIRECT, _16 DIRECT, and _8 DIRECT interpret the offset as a byte address. The other routines treat this offset as a count to be multiplied by four bytes, the number of bytes in the 32-bit connection between the Nios II processor and the system interconnect fabric. The _32DIRECT, _16DIRECT, and _8DIRECT routines are designed to access memory regions, and the other routines are designed to access peripheral registers.

The example below shows how to write a series of half-word values into memory. Because the target addresses are not all on a 32-bit boundary, this code sample uses the `IOWR_16DIRECT` macro.

Example 4-17: Writing Half-Word Locations

```
/* Loop across 100 memory locations, writing 0xdead to */
/* every half word location... */
for(i=0, j=0;i<100;i++, j+=2)
{
IOWR_16DIRECT(MEM_START, j, (unsigned short)0xdead);
}
```

The example below shows how to access a peripheral register. In this case, the write is to a 32-bit boundary address, and the code sample uses the IOWR macro.

Example 4-18: Peripheral Register Access

```
unsigned int control_reg_val = 0;
/* Read current control register value */
control_reg_val = IORD(BAR_BASE_ADDR, CONTROL_REG);

/* Enable "start" bit */
control_reg_val |= 0x01;

/* Write "start" bit to control register to start peripheral */
IOWR(BAR_BASE_ADDR, CONTROL_REG, control_reg_val);
```

Note: Altera recommends that you use the HAL-supplied macros for accessing external peripherals and memory.

Sharing Uncached Memory

If your application must allocate some memory, operate on that memory, and then share the memory region with another peripheral (or processor), use the HAL-supplied `alt_uncached_malloc()` and `alt_uncached_free()` functions. Both of these functions operate on pointers to bypass cached memory.

To share uncached memory between a Nios II processor and a peripheral, perform the following steps:

1. **malloc memory**—Run the `alt_uncached_malloc()` function to claim a block of memory from the heap. If this operation is successful, the function returns a pointer that bypasses the data cache.
2. **Operate on memory**—Have the Nios II processor read or write the memory using the pointer. Your application can perform normal pointer-arithmetic operations on this pointer.
3. **Convert pointer**—Run the `alt_remap_cached()` function to convert the pointer to a memory address that is understood by external peripherals.
4. **Pass pointer**—Pass the converted pointer to the external peripheral to enable it to perform operations on the memory region.

Sharing Memory With Cache Performance Benefits

Another way to share memory between a data-cache enabled Nios II processor and other external peripherals safely without sacrificing processor performance is the delayed data-cache flush method. In this method, the Nios II processor performs operations on memory using standard C or C++ operations until it needs to share this memory with an external peripheral.

Note: Your application can share non-cache-bypassed memory regions with external masters if it runs the `alt_dcache_flush()` function before it allows the external master to operate on the memory.

To implement delayed data-cache flushing, the application image programs the Nios II processor to follow these steps:

1. **Processor operates on memory**—The Nios II processor performs reads and writes to a memory region. These reads and writes are C/C++ pointer or array based accesses or accesses to data structures, variables, or a malloc'ed region of memory.
2. **Processor flushes cache**—After the Nios II processor completes the read and write operations, it calls the `alt_dcache_flush()` instruction with the location and length of the memory region to be flushed. The processor can then signal to the other memory master peripheral to operate on this memory.
3. **Processor operates on memory again**—When the other peripheral has completed its operation, the Nios II processor can operate on the memory once again. Because the data cache was previously flushed, any additional reads or writes update the cache correctly.

The example below shows an implementation of delayed data-cache flushing for memory accesses to a C array of structures. In the example, the Nios II processor initializes one field of each structure in an array, flushes the data cache, signals to another master that it may use the array, waits for the other master to complete operations on the array, and then sums the values the other master is expected to set.

Example 4-19: Data-Cache Flushing With Arrays of Structures

```
struct input foo[100];

for(i=0;i<100;i++)
    foo[i].input = i;
alt_dcache_flush(&foo, sizeof(struct input)*100);
signal_master(&foo);
for(i=0;i<100;i++)
    sum += foo[i].output;
```

The example below shows an implementation of delayed data-cache flushing for memory accesses to a memory region the Nios II processor acquired with `malloc()`.

Example 4-20: Data-Cache Flushing With Memory Acquired Using malloc()

```
char * data = (char*)malloc(sizeof(char) * 1000);

write_operands(data);
alt_dcache_flush(data, sizeof(char) * 1000);
signal_master(data);
result = read_results(data);
free(data);
```

The `alt_dcache_flush_all()` function call flushes the entire data cache, but this function is not efficient. Altera recommends that you flush from the cache only the entries for the memory region that you make available to the other master peripheral.

Handling Exceptions

The HAL infrastructure provides a robust interrupt handling service routine and an API for exception handling. The Nios II processor can handle exceptions caused by hardware interrupts, unimplemented instructions, and software traps.

This section discusses exception handling with the Nios II internal interrupt controller. The Nios II processor also supports an external interrupt controller (EIC), which you can use to prioritize interrupts and make other performance improvements.

For information about the EIC, refer to the Programming Model chapter of the *Nios II Gen2 Processor Reference Handbook*. For information about the exception handler software routines, HAL-provided services, API, and software support for the EIC, refer to the Exception Handling chapter of the *Nios II Gen2 Software Developer's Handbook*.

Consider the following common issues and important points before you use the HAL-provided exception handler:

- **Prioritization of interrupts**—The Nios II processor does not prioritize its 32 interrupt vectors, but the HAL exception handler assigns higher priority to lower numbered interrupts. You must modify the interrupt request (IRQ) prioritization of your peripherals in Qsys.
- **Nesting of interrupts**—The HAL infrastructure allows interrupts to be nested— higher priority interrupts can preempt processor control from an exception handler that is servicing a lower priority interrupt. However, Altera recommends that you not nest your interrupts because of the associated performance penalty.
- **Exception handler environment**—When creating your exception handler, you must ensure that the handler does not run interrupt-dependent functions and services, because this can cause deadlock. For example, an exception handler should not call the IRQ-driven version of the `printf()` function.
- **VIC block**—Vector interrupt controller block provides an interface to the interrupts in your system. The VIC offers high-performance, low-latency interrupt handling. The VIC prioritizes interrupts in hardware and outputs information about the highest-priority pending interrupt. For more information, refer to the "Vectored Interrupt Controller Core" chapter of the *Embedded Peripheral IP User Guide*.

Related Information

- **Embedded Peripherals IP User Guide**
- **Programming Model**
- **Exception Handling**

Modifying the Exception Handler

In some very special cases, you may wish to modify the existing HAL exception handler routine or to insert your own interrupt handler for the Nios II processor. However, in most cases you need not modify the interrupt handler routines for the Nios II processor for your software application.

Consider the following common issues and important points before you modify or replace the HAL-provided exception handler:

- **Interrupt vector address**—The interrupt vector address for each Nios II processor is set during compilation of the FPGA design. You can modify it during hardware configuration in Qsys.
- **Modifying the exception handler**—The HAL-provided exception handler is fairly robust, reliable, and efficient. Modifying the exception handler could break the HAL-supplied interrupt handling API, and cause problems in the device drivers for other peripherals that use interrupts, such as the UART and the JTAG UART.

You may wish to modify the behavior of the exception handler to increase overall performance. For guidelines for increasing the exception handler's performance, refer to "Accelerating Interrupt Service Routines".

Related Information

Accelerating Interrupt Service Routines on page 7-18

Linking Applications

This section discusses how the Nios II software development tools create a default linker script, what this script does, and how to override its default behavior. The section also includes instructions to control some common linker behavior, and descriptions of the circumstances in which you may need them.

Background

When you generate your project, the Nios II Software Build Tools generate two linker-related files, **linker.x** and **linker.h. linker.x** is the linker command file that the generated application's makefile uses to create the **.elf** binary file. All linker setting modifications you make to the HAL BSP project affect the contents of these two files.

Linker Sections and Application Configuration

Every Nios II application contains `.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack` sections. Additional sections can be added to the **.elf** file to hold custom code and data.

These sections are placed in named memory regions, defined to correspond with physical memory devices and addresses. By default, these sections are automatically generated by the HAL. However, you can control them for a particular application.

HAL Linking Behavior

This section describes the default linking behavior of the BSP generation tools and how to control the linking explicitly.

Default BSP Linking

During BSP configuration, the tools perform the following steps automatically:

1. **Assign memory region names**—Assign a name to each system memory device, and add each name to the linker file as a memory region.
2. **Find largest memory**—Identify the largest read-and-write memory region in the linker file.
3. **Assign sections**—Place the default sections (`.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack`) in the memory region identified in the previous step.
4. **Write files**—Write the linker.x and linker.h files.

Usually, this section allocation scheme works during the software development process, because the application is guaranteed to function if the memory is large enough.

The rules for the HAL default linking behavior are contained in the Altera-generated Tcl scripts **bsp-set-defaults.tcl** and **bsp-linker-utils.tcl** found in the `<Nios II EDS install dir>/sdk2/bin` directory. These scripts are called by the **nios2-bsp-create-settings** configuration application. Do not modify these scripts directly.

User-Controlled BSP Linking

You can manage the default linking behavior in the Linker Script tab of the Nios II BSP Editor. You can manipulate the linker script in the following ways:

- Add a memory region—Maps a memory region name to a physical memory device.
- Add a section mapping—Maps a section name to a memory region. The Nios II BSP Editor allows you to view the memory map before and after making changes.

For more information about the linker-related BSP configuration commands, refer to "Using the BSP Editor" in the Getting Started with the Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Getting Started with the Graphical User Interface

Nios II MPU Usage

The Nios II MPU Usage section covers the basic features of the Nios II processor's optional memory protection unit (MPU), describing how to use it without the support of an operating system (OS). When the Nios II MPU is enabled and properly configured, it monitors all processor data and instruction accesses and triggers exceptions when illegal accesses are attempted.

Also included are two design examples, with notes about how they work. These examples walk you through making use of the Nios II processor's MPU in an environment based on the Altera hardware abstraction layer (HAL), without an OS. One of the examples uses the MPU to detect the following three issues commonly seen when debugging embedded systems:

- Stack overflow
- Null pointer
- Wild pointer

Note: Do not confuse the MPU with the Nios II memory management unit (MMU). The MPU does not provide memory mapping or management.

After you have studied the code and understand the design examples described in this section, you have the skills to use the Nios II MPU successfully in your HAL-based design. These examples illustrate the basics of how to use mpubase and mpuacc to configure your MPU prior to enabling it.

Requirements

To use this section effectively, you need to be familiar with the following topics:

- The basic purpose and architecture of the Nios II MPU

  Note: For a detailed description of the Nios II MPU, refer to "Memory Protection Unit" in the Programming Model chapter of the *Nios II Processor Reference Handbook*.

To work with this section's design examples and software examples, you need the following items:

- The Nios II Embedded Evaluation Kit (NEEK), Cyclone® III Edition

  Note: The design examples use only on-chip hardware resources. Therefore, it is easy to port the designs to a different hardware platform if necessary.
- Quartus Prime software.
- Nios II Embedded Design Suite (EDS).
- The design example archive file, **an540_91.zip**. This file is available on the Literature: Nios II Processor page of the Altera website.

  Unzip **an540_91.zip** to a working directory on your computer. We refer to this directory throughout this section as <*design examples*>. Be sure to preserve the directory structure of the extracted software archive. Extraction creates a directory structure tree under <design examples> with the following subdirectories:

  - `MPU_Design_limit/software_examples/app/mpu_basic`
  - `MPU_Design_limit/software_examples/app/mpu_exc_detection`
  - `MPU_Design_limit/software_examples/bsp/mpu_example_bsp`
  - `MPU_Design_msk/software_examples/app/mpu_basic`
  - `MPU_Design_msk/software_examples/app/mpu_exc_detection`
  - `MPU_Design_msk/software_examples/bsp/mpu_example_bsp`

  Note: The working directory name you choose must not contain any spaces.

Note: After extracting **an540_91.zip**, refer to <*design examples*>**/ReadMe.txt** for a list of any required software patches or other updated information. If a patch is required, install it according to the instructions in **ReadMe.txt**.

Related Information

- **Programming Model**
- **Literature: Nios II Processor**

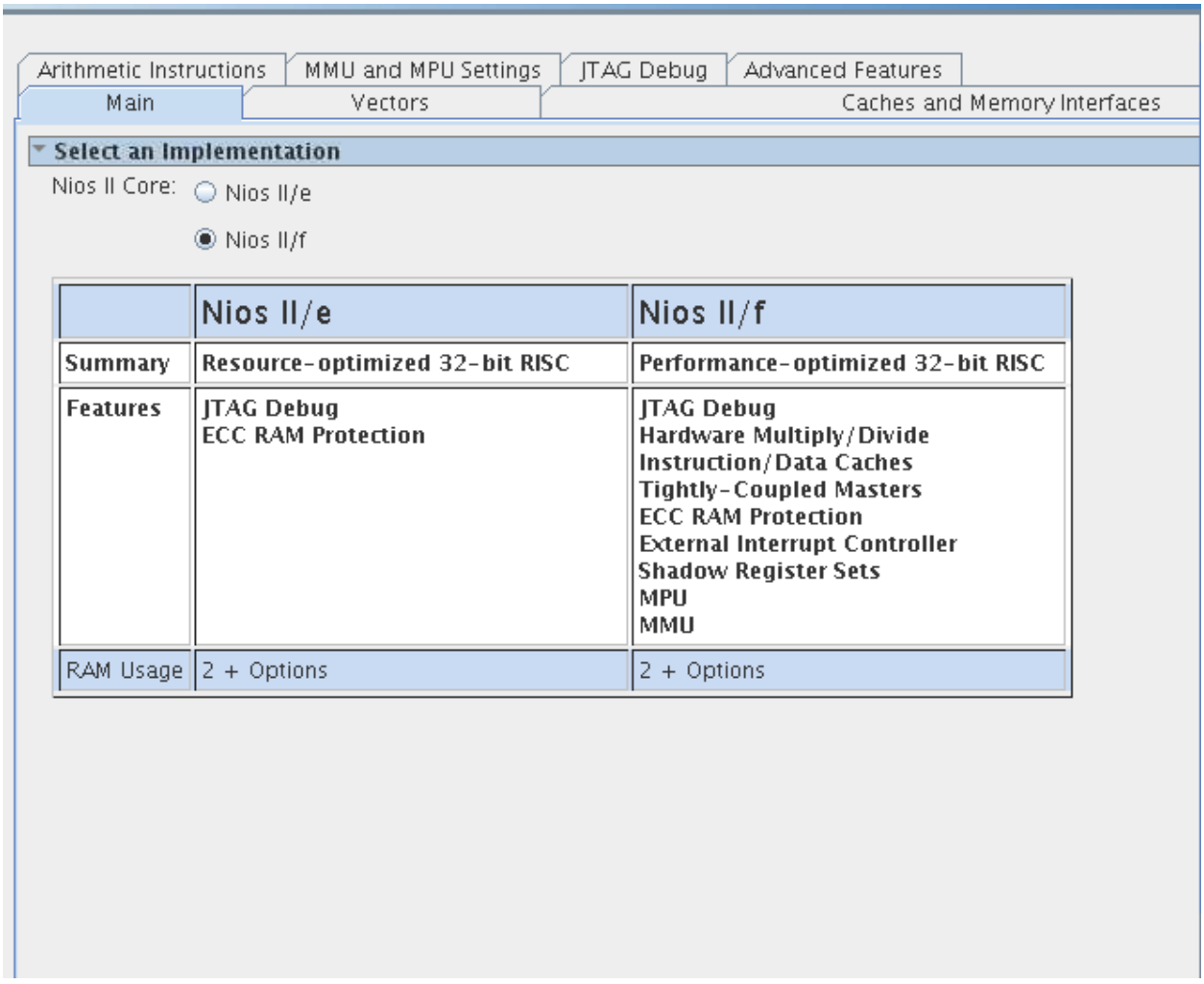
General Usage

This section describes the process of configuring the Nios II MPU hardware and writing software to support it.

Adding the MPU Hardware

To add an MPU to your system, you must use a Nios II/f core. In Qsys, enable the MPU by turning on **Include MPU** in the **Core Nios II** tab of the Nios II parameter editor interface, as shown in below.

Figure 4-7: Enabling the MPU in the Nios II/f Processor Core



Use the MMU and MPU Settings tab, as shown below, to configure the MPU.

Figure 4-8: MMU and MPU Settings Tab

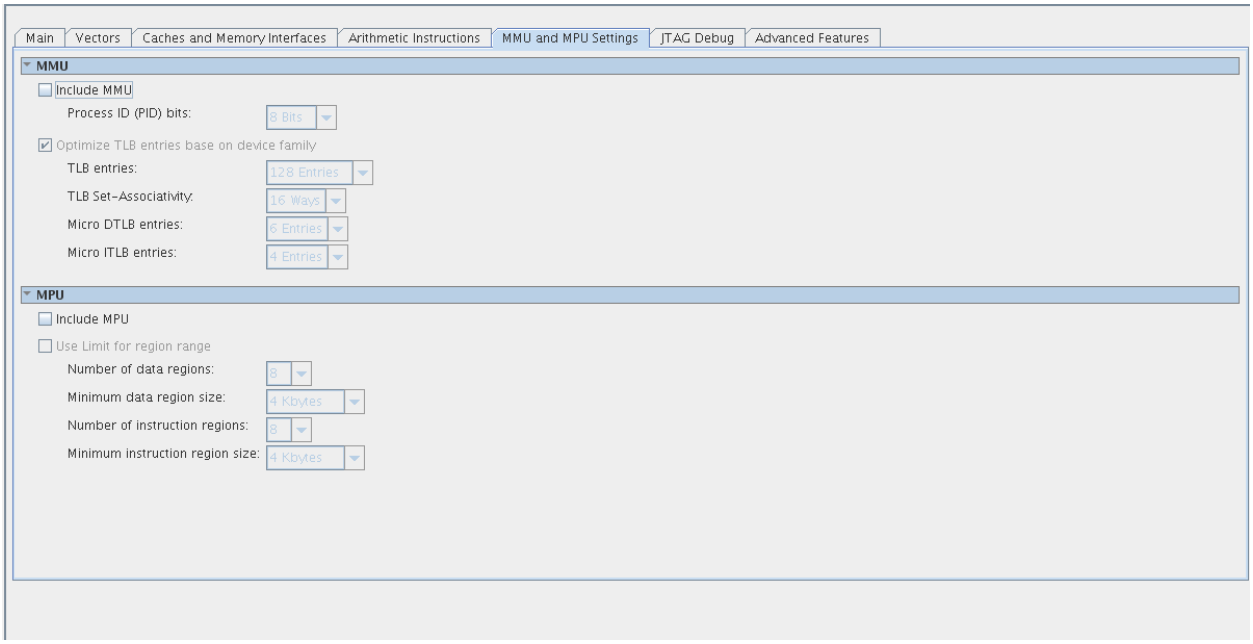


Table 4-4: MPU Configuration Options

Option	Allowed Values	Default Value
Use Limit for Region Range	Off or On	Off
Number of Data Regions	2—32	8
Minimum Data Region Size	256 bytes—1 MB	4 KB
Number of Instruction Regions	2—32	8
Minimum Instruction Region Size	256 bytes—1 MB	4 KB

You can configure the MPU to define the size of its memory regions in either of the following ways:

- Define region size by specifying an address mask
- Define region size by specifying the end address

By default, the MPU defines region sizes with an address mask. To define region sizes with an end address, turn on **Use Limit for Region Range**. For detailed information about the two methods of specifying region size, refer to "MPU Register Details" section.

The minimum region size is crucial to understanding MPU run-time configuration. The minimum region size, <*min_region*>, specifies the granularity of the MPU memory map. The size of any particular memory region must be an integer multiple of <*min_region*>.

Most of the MPU parameters controlled by software are based on the minimum region size. You can specify separate values of <*min_region*> for data and instruction regions.

Note: For simplicity, this section's design examples have <*min_region*> = 64 for both data and instruction regions.

Related Information

MPU Register Details on page 4-65

Writing Software for the MPU

This section describes the process of writing software to configure and manage the Nios II MPU.

MPU Programming Guidelines

Software is responsible for enabling and configuring the MPU as well as maintaining MPU region information. In a single-threaded operating environment (such as the Altera HAL), use a global data structure to store the MPU region information.

The Nios II MPU must be disabled before software attempts to configure it.

Software normally initializes the MPU after reset. If it is necessary to change MPU regions or region permissions after reset, software also reinitializes the MPU.

Every region supported by the MPU must be either configured or disabled before allowing application code to execute. Leaving a region enabled and unconfigured results in undefined behavior. For details about how to disable an MPU region, refer to "Defining Regions with mpubase and mpuacc" section.

Depending on the complexity of your software, you might need to define several MPU configurations, each with a different set of regions or region permissions. This technique is typically used by an operating system. For details, refer to "Operating Systems and the MPU" section.

Related Information

- **Defining Regions with mpubase and mpuacc** on page 4-68
- **Operating Systems and the MPU** on page 4-64

Operating Systems and the MPU

Even if you are not using an operating system, it is helpful to understand the techniques that an OS uses to manage an MPU.

When an operating system uses an MPU, it typically defines two or more MPU configurations. One configuration defines the permissions that the MPU applies to operating system or kernel level accesses. One or more configurations define the permissions available to user or application processes. The OS might also define additional configurations for non-user purposes. For example, there might be a special factory task that can modify system-critical information like product serial numbers or media access control (MAC) addresses in flash or other nonvolatile memory. Such a task is likely to need a special set of memory and device permissions.

The operating system disables the MPU, reconfigures it, and then re-enables it whenever the processor needs to run in a different MPU configuration. For example, the OS might need to change MPU configurations upon the following types of events:

- Exception
- Return from exception
- Operating system call
- Return from operating system call

The exact circumstances under which MPU reconfiguration is required depends on the OS implementation and settings.

MPU Register Details

This section describes the register maps, the meanings of the register fields, and how the register fields are used.

When you initialize the MPU you use two registers: `mpubase` and `mpuacc`.

Register mpubase Usage

Table 4-5: mpubase Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BASE(2)																										INDEX(3)					D

Table 4-6: mpubase Control Register Field Descriptions

Field	Description	Access	Reset
BASE	BASE represents the base memory address of the region identified by the INDEX and D fields.	Read/Write	0
INDEX	INDEX is the region index number.	Read/Write	0
D	D is the region access bit. When D = 1, INDEX refers to a data region. When D = 0, INDEX refers to an instruction region.	Read/Write	0

You specify an MPU region by writing a value representing the region's base address to the BASE field, a unique index to the INDEX field, and the region type (data or instruction) to field D.

The BASE field represents the region's base address, in the form described by the equation below. The BASE field can only represent addresses aligned to an integer multiple of <*min_region*>. For example, if the minimum region size is 16 kilobytes (KB), regions can be located at addresses such as 0x0, 0x4000, 0x8000, ...

Figure 4-9: Base Address Computation

BASE = <base address>/<min_region>

For example, if the region starts at 0x1000 and the minimum region size is 64 bytes, set the BASE field to 0x40, which is 0x1000/64.

The INDEX field provides a unique identifier for the region. INDEX also specifies the priority of the region. The lower the index value, the higher the region's priority.

Use the D field to specify the region type: data or instruction.

Register mpuacc Usage

mpuacc has two possible layouts, depending on the Qsys generation-time option Use limit for region range, as described in "Adding the MPU Hardware" section. This option controls whether the mpuacc register contains a MASK or LIMIT field. The table below shows the layout of the mpuacc register with the MASK field.

(2) This field size is variable. Unused upper bits and unused lower bits must be written as zero.
(3) This field size is variable. Unused upper bits must be written as zero.

Table 4-7: mpuacc Control Register Fields for MASK Variation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	MASK																									C	PERM			RD	WR

Table 4-8: mpuacc Control Register Fields for LIMIT Variation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	LIMIT(4)																									C	PERM			RD	WR

Table 4-9: mpuacc Control Register Field Descriptions

Field	Description	Access	Reset
MASK(5)	MASK specifies the size of the region.	Read/Write	0
LIMIT(5)	LIMIT specifies the upper address limit of the region.	Read/Write	0
C	C is the data cacheable flag. C only applies to MPU data regions and determines the default cacheability of a data region. When C = 0, the data region is uncacheable. When C = 1, the data region is cacheable.	Read/Write	0
PERM	PERM specifies the access permissions for the region.	Read/Write	0
RD	RD is the read region flag. When RD = 1, wrctl instructions to the mpuacc register perform a read operation.	Write	0
WR	WR is the write region flag. When WR = 1, wrctl instructions to the mpuacc register perform a write operation.	Write	0

If the mpuacc register is configured with the MASK field, the MASK field represents the size of your region. The value of MASK is defined in the equation below.

Figure 4-10: Computing Region Mask

$$MASK = 0x1FFFFFF << \log_2 (<region\_size> >> 6)$$

The table below lists every possible MASK value for an MPU configured with a 64-byte minimum region size.

Table 4-10: MASK Encodings for 64-byte Minimum Region

MASK Encoding	Region Size
0x1FFFFFF	64 bytes
0x1FFFFFE	128 bytes

(4) This field size is variable. Unused upper bits and unused lower bits must be written as zero.
(5) The MASK and LIMIT fields are mutually exclusive.

MASK Encoding	Region Size
0x1FFFFFC	256 bytes
0x1FFFFF8	512 bytes
0x1FFFFF0	1 KByte
0x1FFFFE0	2 KB
0x1FFFFC0	4 KB
0x1FFFF80	8 KB
0x1FFFF00	16 KB
0x1FFFE00	32 KB
0x1FFFC00	64 KB
0x1FFF800	128 KB
0x1FFF000	256 KB
0x1FFE000	512 KB
0x1FFC000	1 MB
0x1FF8000	2 MB
0x1FF0000	4 MB
0x1FE0000	8 MB
0x1FC0000	16 MB
0x1F80000	32 MB
0x1F00000	64 MB
0x1E00000	128 MB
0x1C00000	256 MB
0x1800000	512 MB
0x1000000	1 GB
0x0000000	2 GB

If the `mpuacc` register is configured with the `LIMIT` field, `LIMIT` represents the address immediately following the upper end of your region. For example, suppose the MPU's minimum region size is 64 bytes, and you need to set up the following region:

- The region starts at 0x1000
- The region ends at 0x1FFF

To set up the desired region, configure `mpubase.BASE` and `mpuacc.LIMIT` as shown in the following list:

- Set `mpubase.BASE` to 0x40, which is 0x1000/64
- Set `mpuacc.LIMIT` to 0x80, which is 0x2000/64

Use the `C` field to specify whether a data region is to be cached. Usually, you set `C` for memory regions and clear it for regions representing registers or general-purpose memory-mapped I/O.

The PERM field defines the permissions for the region, as shown in the two tables below.

Table 4-11: Instruction Region Permission Encodings

PERM Encoding	Supervisor Permissions	User Permissions
000	Noe	None
001	Execute	None
010	Execute	Execute

Table 4-12: Data Region Permission Encodings

PERM Encoding	Supervisor Permissions	User Permissions
000	None	None
001	Read	None
010	Read	Read
100	Read/Write	None
101	Read/Write	Read
110	Read/Write	Read/Write

Related Information

Adding the MPU Hardware on page 4-61

Defining Regions with mpubase and mpuacc

The `mpubase` register works in conjunction with the `mpuacc` register to set and retrieve MPU region information. Use the `RD` and `WR` fields of `mpuacc` to instruct the MPU to perform an MPU region read or write, as shown in the following list:

- Set `mpuacc.RD` = 1 to perform an MPU region read operation.
- Set `mpuacc.WR` = 1 to perform an MPU region write operation.

Note: Simultaneously setting both the `RD` and `WR` fields to 1 results in undefined behavior.

An MPU region must be disabled if it is not in use. To disable a region, software sets up the following conditions:

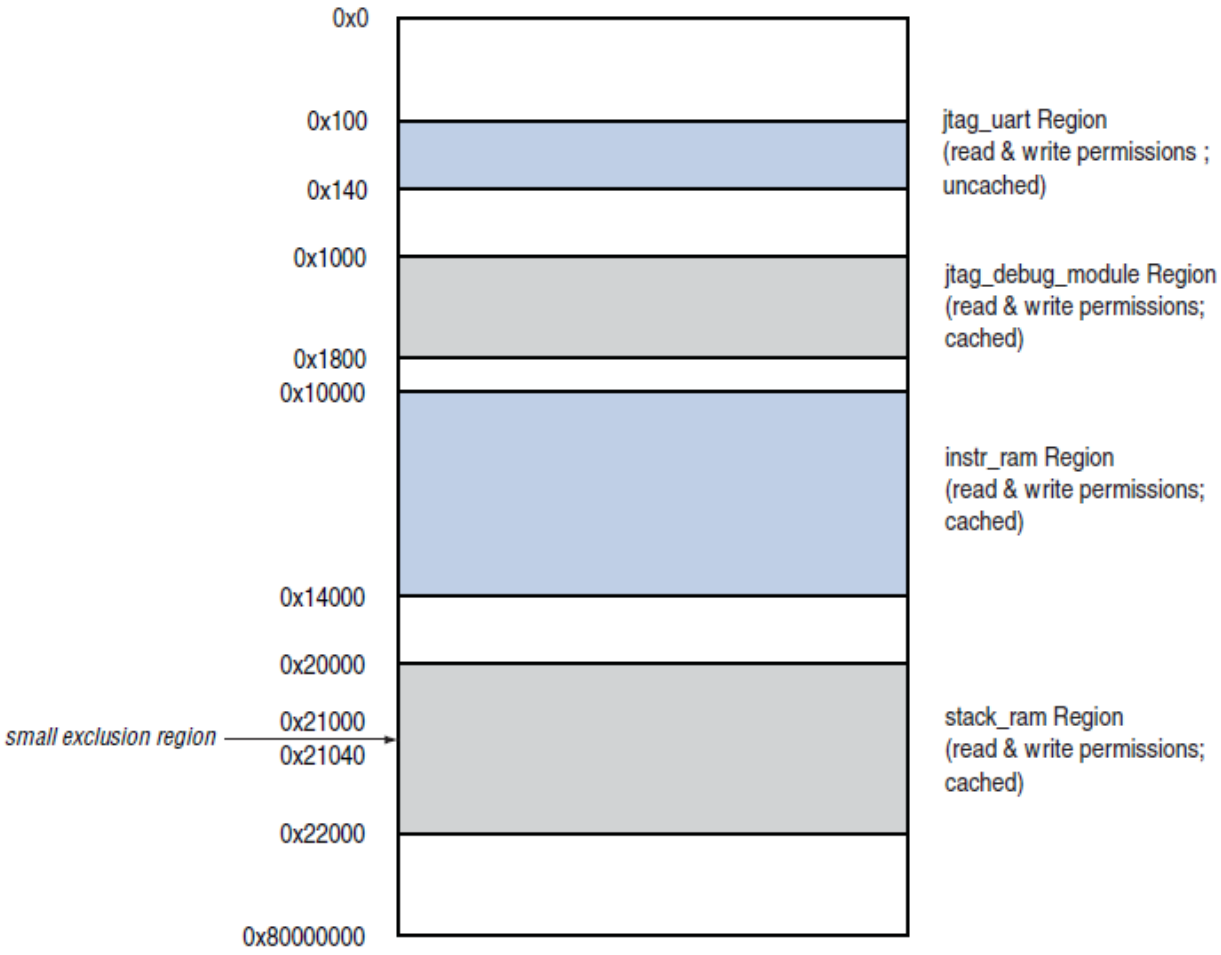
- `mpubase.BASE` is any nonzero value.
- If the MPU is configured to define region size by mask, `mpuacc.MASK` represents 0x80000000, which is 2^{31} (the size of the Nios II address space). For example, if the minimum region size is 64, or 0x40 bytes, `mpuacc.MASK` is 0x80000000 / 0x40, or 0x20000000.
- If the MPU is configured to define region size by limit, `mpuacc.LIMIT = 0`.

Region Layout Considerations

This section describes how to select MPU region locations and sizes to make the most effective use of the MPU. For information about the mechanics of setting up MPU regions, refer to "MPU Register Details" section.

Each region size must be an integer power of two. You must ensure that each region is aligned to an address that is an integer multiple of its size.

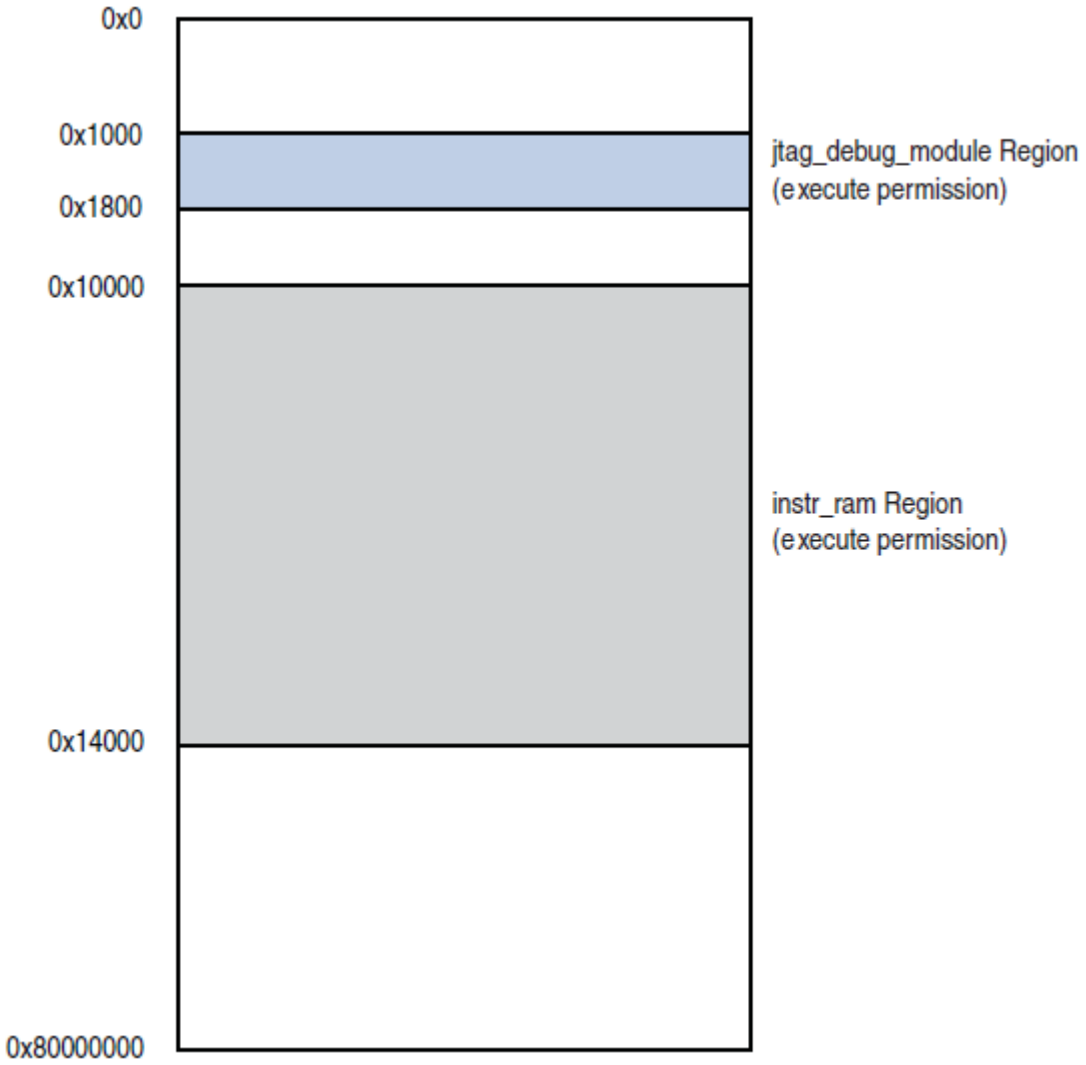
Figure 4-11: MPU Data Region Example (Addresses not to scale)



A low-priority exclusion region spans the entire 2 GB address space from 0x0 to 0x80000000.

Regions can overlap. For example, you can place a higher-priority region inside a lower-priority region. region[3] in mpu_utils.c illustrates this technique, creating a small exclusion region from 0x21000 to 0x21040, as shown in Figure 3. Any access to addresses in the 0x21000 to 0x21040 range is controlled by the exclusion region rather than the stack_ram region (region[4]), because the exclusion region has the higher priority.

Figure 4-12: MPU Instruction Region Example (Addresses not to scale)



A low-priority exclusion region spans the entire 2 GB address space from 0x0 to 0x80000000.

Related Information

MPU Register Details on page 4-65

Flow Summary

In a Nios II system with an MPU, whenever MPU initialization or reinitialization is required, the software is responsible for the following tasks:

1. Ensure that the MPU is disabled.

Note: At system reset, the MPU is disabled by default. At other times, software must disable the MPU before reconfiguring regions.

2. Initialize and configure the MPU with region information.
3. Enable the MPU prior to executing task-specific or single-threaded application code.

Nios II MPU Design Examples

The design examples accompanying this section illustrate the use of the Nios II MPU in a single-threaded environment, such as the Altera HAL.

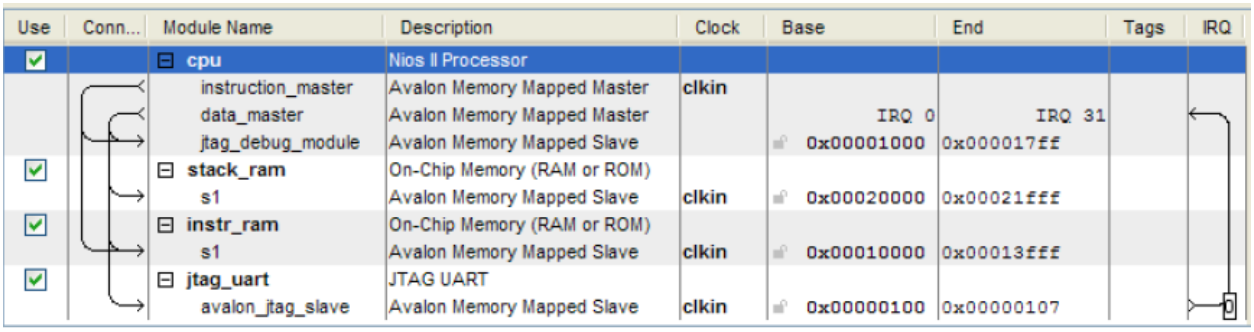
Example Hardware

The simple hardware designs, emphasizing MPU usage, are easily portable to other hardware platforms. There are two design examples, both targeting the NEEK. In one, the MPU specifies region sizes by mask, and in the other the MPU specifies region sizes by limit. Aside from this detail of MPU instantiation, the two designs are identical.

The address map is designed to make MPU configuration very straightforward. For instance, the **instr_ram** and **stack_ram** memories reside on valid region boundaries, and the JTAG UART base address is unique and aligned to a valid region boundary, as illustrated in Figure 4-11.

The figure below illustrates one of the design examples as it appears in Qsys. The hardware addresses fall on valid MPU region boundaries. While this constraint is not required, it is more convenient for the software engineer.

Figure 4-13: MPU Example Hardware System



Software

The design files accompanying this section include the following example software projects:

- **mpu_basic**—Configures the MPU with several data and instruction regions, and prints a simple message.
- **mpu_exc_detection**—Configures the MPU with the same data instruction regions as in **mpu_basic**, and sets up an exception handler to detect the following conditions:
  - Null pointer
  - Wild pointer
  - Stack overflow

The software examples in each subdirectory are identical. The code is written to detect the whether the MPU is configured for mask or limit region sizes, and to behave appropriately.

The **mpu_exc_detection** example detects stack overflow by creating a small high-priority exclusion data region in the middle of a larger data region where both the stack and the heap reside. Whenever the stack grows downwards or the heap grows upwards into this exclusion region, the MPU triggers an exception and the software detects it.

The **mpu_exc_detection** example detects null pointer usage by making sure that no regions include offset 0x0. The example system is designed such that no components (memory or otherwise) are located at this offset. If software attempts to access address 0x0, the MPU triggers an exception, allowing the software to recover. If you ensure that memories are preinitialized to zero, null pointer detection helps protects against uninitialized data access.

The **mpu_exc_detection** example detects wild pointer usage by creating very large low-priority exclusion regions covering the majority of the memory map. In this way, if the Nios II processor attempts to access an address outside of valid memory and peripheral I/O address space, the MPU triggers an exception and software can detect it.

Both of these software examples use the MPU utility functions and macros in **mpu_utils.c** and **mpu_utils.h**. In both examples, initialization and reinitialization are handled by two functions: one for data regions, and one for instruction regions. In most real-world systems, a single function is sufficient to handle initialization and reinitialization for both types of regions.

MPU Utilities

You can find helpful MPU utility functions and macros in the **mpu_utils.c** and **mpu_utils.h** files in each software example. The following functions are the most important for you to understand:

- `nios2_mpu_data_init()`—A system-specific function. In your own code, write an equivalent function to specify the MPU data regions in your design.
- `nios2_mpu_inst_init()`—A system-specific function. In your own code, write an equivalent function to specify the MPU instruction regions in your design.
- `nios2_mpu_load_region()`—Configures an MPU region with specific parameters.
- `nios2_mpu_enable()`—Enables the entire MPU.
- `nios2_mpu_disable()`—Disables the entire MPU.

Each utility function makes use of the `Nios2MPURegion` data structure shown in the example below.

Example 4-21: Nios2MPURegion Data Structure

```
typedef struct
{
    unsigned int base;
    unsigned int index;
    unsigned int mask;
    unsigned int c;

    unsigned int perm;
} Nios2MPURegion;
```

The example below shows `nios2_mpu_inst_init()` for the **mpu_basic** software example. The constants `NIOS2_MPU_NUM_INST_REGIONS` and `NIOS2_MPU_REGION_USES_LIMIT` are defined in **system.h**.

In Example 4-22, `region[0]` grants execution access to the **instr_ram** memory in both user and supervisor modes, as shown in Figure 4-12. `region[1]` grants execution access to the break and trace memory (starting at 0x1000) in both modes. The other two MPU instruction regions grant no execution permissions to the entire Nios II address space. Because their priorities, 2 and 3, are lower than the first

two regions, the code stored in the `instr_ram` runs, and the break and trace features work correctly. However, if code attempts to execute outside those regions, the MPU triggers an exception.

The final statement in `nios2_mpu_inst_init()` calls `nios2_mpu_load_region()` to configure the region with the information contained in the structure.

Example 4-22: nios2_mpu_inst_init() in the mpu_basic Software Example

```
void nios2_mpu_inst_init()
{
    unsigned int mask;
    Nios2MPURegion region[NIOS2_MPU_NUM_INST_REGIONS];

    //Main instruction region.
    region[0].index = 0;

    region[0].base = 0x400; // Byte Address 0x10000
#ifdef NIOS2_MPU_REGION_USES_LIMIT
    region[0].mask = 0x500; // Byte Address 0x14000
#else
    region[0].mask = 0x1ffff00;
#endif
    region[0].c = 1;
    region[0].perm = MPU_INST_PERM_SUPER_EXEC_USER_EXEC;

    //Instruction region for break address.
    region[1].index = 1;
    region[1].base = 0x40; // Byte Address 0x1000
#ifdef NIOS2_MPU_REGION_USES_LIMIT
    region[1].mask = 0x60; // Byte Address 0x1800
#else
    region[1].mask = 0x1ffffe0;
#endif
    region[1].c = 1;
    region[1].perm = MPU_INST_PERM_SUPER_EXEC_USER_EXEC;

    //Rest of the regions are maximally sized and permissive.
#ifdef NIOS2_MPU_REGION_USES_LIMIT
    mask = 0x2000000;
#else
    mask = 0x0;
#endif
    unsigned int num_of_region = NIOS2_MPU_NUM_INST_REGIONS;
    unsigned int index;
    for (index = 2; index < num_of_region; index++){
        region[index].base = 0x0;
        region[index].index = index;
        region[index].mask = mask;
        region[index].c = 0;
        region[index].perm = MPU_INST_PERM_SUPER_NONE_USER_NONE;
    }
    nios2_mpu_load_region(region, num_of_region, 0);
}
```

The example below shows the function prototype for `nios2_mpu_load_region()`.

Example 4-23: nios2_mpu_load_region()

```
void nios2_mpu_load_region (
    Nios2MPURegion region[],
```

```
    unsigned int num_of_region,
    unsigned int d);
```

The following list shows the arguments to `nios2_mpu_load_region()`:

- `Nios2MPURegion`—An array of data structures, each representing an MPU region
- `num_of_region`—The number of regions
- `d`—The region type (instruction or data)

`nios2_mpu_load_region()` configures the MPU according to the arguments passed by the calling function.

The MPU is disabled by default at system restart. After the MPU is configured, the example uses `nios2_mpu_enable()` and `nios2_mpu_disable()` to enable and disable the MPU. Whenever you reconfigure the MPU, you must first disable it, and re-enable it after configuring.

The software examples accompanying this section are commented to help you understand how each example works. Most of the complexity of managing the MPU and its regions is embodied in the MPU utility functions and macros in **mpu_utils.c** and **mpu_utils.h**, allowing you to focus on the top-level software flow.

Building the Software

To create and build a software example, execute the following steps:

1. Identify the directory containing the software example that you want to run, based on the hardware example that you want to use. For example, to run the **mpu_basic** software example on the **MPU_Design_limit** hardware design example, the directory is <*design examples*>/**MPU_Design_limit/software_examples/app/ mpu_basic**.
2. Use one of the following methods to open the Nios II Command Shell:
   - In the Windows operating system, on the Start menu, point to **Programs > Altera > Nios II EDS** <*version*>, and click **Nios II** <*version*> **Command Shell**.
   - In the Linux operating system, in a command shell, execute the following commands:

     ```
     cd $SOPC_KIT_NIOS2
     ./sdk_shell
     ```
3. Change directories to the software example directory identified in Step 1.
4. Type the following command:

   ```
   ./create-this-app
   ```
5. After the projects are generated and built, configure your board with the hardware image and run the software with the following commands:

   ```
   nios2-configure-sof -C ../../../
   nios2-download -g <example>.elf && nios2-terminal
   ```

   Each software example displays information on the screen. The output from the **mpu_basic** example resembles the example below.

Example 4-24: mpu_basic Console Output

```
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Pausing target processor: OK
```

```
Initializing CPU cache (if present)
OK
Downloaded 3KB in 0.0s
Verified OK
Starting processor at address 0x00010020
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Hello from a simple MPU-Enabled Nios II System!.
val1 = 0xfeedface, val2 = 0xfeedface, val3 = 0x@.
```

The output from the **mpu_exc_detection** example resembles **mpu_exc_detection Console Output** on page 4-75.

mpu_exc_detection Console Output

Example 4-25: mpu_exc_detection Console Output

```
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 5KB in 0.0s
Verified OK
Starting processor at address 0x00010110
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Hello from a simple MPU-Enabled Nios II System!.
    Starting some exceptions tests.
=====
MPU NULL data pointer test.
MPU NULL data pointer test passed!
MPU wild pointer test.
MPU wild pointer test passed!
MPU stack overflow test.
MPU stack overflow test passed!
=====
    Exception Tests ended.
Now exiting program.
```

For further details, refer to the source code and the <*design examples*>/**ReadMe.txt** file accompanying the examples.

If the software example appears to hang, verify that you have configured your board with the correct **.sof**.

Document Revision History

Table 4-13: Software System Design with a Nios II Processor Chapter Revision History

Date	Version	Changes
December 2016	2016.12.19	Initial release.

Nios II Configuration and Booting Solutions

5

2016.12.19

ED_HANDBOOK

The Nios II processor is a soft processor that supports all System on Chip (SoC) and Field Programmable Gate Array (FPGA) families.

This chapter describes the various boot or software execution options available with the Nios II processor. You can configure the Nios II processor to boot and execute software from different memory locations. The boot memory could be the Common Flash Interface (CFI) flash, User Flash Memory (UFM) flash in MAX 10, Altera Serial Flash (EPCS)/Altera Quad Serial Flash (EPCQ) configuration device, Quad Serial Parallel Interface (QSPI) flash or On-Chip RAM (OCRAM).

Configuration

Many FPGA configuration options are available to you. The two most commonly used options configure the FPGA from flash memory. One option uses a CPLD and a CFI flash device to configure the FPGA, and the other uses a serial flash EPCS configuration device. The Nios II development kits use these two configuration options by default.

Choose the first option, which uses a CPLD and a CFI-compliant flash memory, in the following cases:

- Your FPGA is large
- You must configure multiple FPGAs
- You require a large amount of flash memory for software storage
- Your design requires multiple FPGA hardware images (safe factory images and user images) or multiple software images

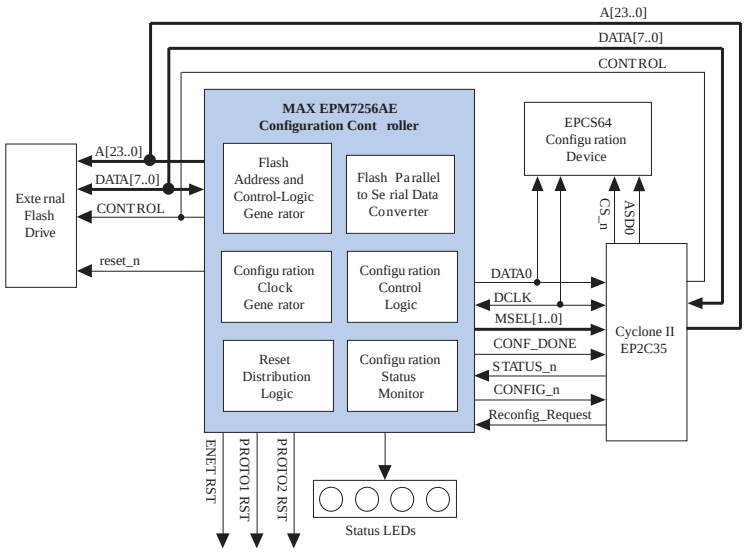
EPCS configuration devices are often used to configure small, single-FPGA systems.

Note: The default Nios II boot loader does not support multiple FPGA images in EPCS devices.

For help in configuring your particular device, refer to the device family information on the Altera Products page of the Altera website.

The figure below shows the block diagram of the configuration controller used on the Nios II Development Kit, Cyclone II Edition. This controller design is used on older development kits, and is a good starting point for your design.

Figure 5-1: Configuration Controller for Cyclone II Devices



For more information about controller designs, refer to AN346: Using the Nios II Configuration Controller Reference Designs.

Related Information

Using the Nios II Configuration Controller Reference Designs

Booting

Many Nios II booting options are available. The following options are the most commonly used:

- Boot from CFI Flash
- Boot from EPCS
- Boot from on-chip RAM

The default boot loader that is included in the Nios II EDS supports boot from CFI flash memory and from EPCS flash memory. If you use an on-chip RAM that supports initialization, such as the M4K and M9K types of RAM, you can boot from the on-chip RAM without a boot loader.

For additional information about Nios II boot methodologies, refer to *AN458: Alternative Nios II Boot Methods*.

Related Information

- **Alternative Nios II Boot Methods**
- **Generic Nios II Booting Methods User Guide**
- **AN736: Nios II Processor Booting From Altera Serial Flash (EPCQ)**
- **AN730: Nios II Processor Booting Methods in MAX 10 FPGA Devices**

Application Boot Loading and Programming System Memory

Most Nios II systems require some method to configure the hardware and software images in system memory before the processor can begin executing your application program. This section describes various possible memory topologies for your system (both volatile and non-volatile), their use, their requirements, and their configuration. The Nios II software application requires a boot loader application to configure the system memory if the system software is stored in flash memory, but is configured to run from volatile memory. If the Nios II processor is running from flash memory—the `.text` section is in flash memory—a copy routine, rather than a boot loader, loads the other program sections to volatile memory. In some cases, such as when your system application occupies internal FPGA memory, or is preloaded into external memory by another processor, no configuration of the system memory is required.

Related Information

- **Nios II Processor Booting From Altera Serial Flash (EPCQ)**
- **EPCS to EPCQ Migration Guideline**
- **AN736: Nios II Processor Booting Methods in MAX 10 FPGA Devices**
- **Alternative Nios II Boot Methods**

Default BSP Boot Loading Configuration

The **nios2-bsp** script determines whether the system requires a boot loader and whether to enable the copying of the default sections.

By default, the **nios2-bsp** script makes these decisions using the following rules:

- **Boot loader**—The **nios2-bsp** script assumes that a boot loader is being used if the following conditions are met:
  - The Nios II processor's reset address is not in the `.text` section.
  - The Nios II processor's reset address is in flash memory.
- **Copying default sections**—The **nios2-bsp** script enables the copying of the default volatile sections if the Nios II processor's reset address is set to an address in the .text section.

If the default boot loader behavior is appropriate for your system, you do not need to intervene in the boot loading process.

Boot Configuration Options

You can modify the default nios2-bsp script behavior for application loading by using the following settings:

- `hal.linker.allow_code_at_reset`
- `hal.linker.enable_alt_load`
- `hal.linker.enable_alt_load_copy_rwdata`
- `hal.linker.enable_alt_load_copy_exceptions`
- `hal.linker.enable_alt_load_copy_rodata`

If you enable these settings, you can override the BSP's default behavior for boot loading. You can modify the application loading behavior in the Settings tab of the Nios II BSP Editor.

Alternatively, you can list the settings in a Tcl script that you import to the BSP Editor.

For information about using an imported Tcl script, refer to "Using Tcl Scripts with the Nios II BSP Editor".

These settings are created in the settings.bsp configuration file whether or not you override the default BSP generation behavior. However, you may override their default values.

For more information about BSP configuration settings, refer to the "Settings Managed by the Software Build Tools" section in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*. For more information about boot loading options and for advanced boot loader examples, refer to *AN458: Alternative Nios II Boot Methods*.

Related Information

- **Nios II Software Build Tools Reference**
- **Alternative Nios II Boot Methods**
- **Using Tcl Scripts with the Nios II BSP Editor** on page 4-31

Booting and Running From Flash Memory

If your program is loaded in and runs from flash memory, the application's .text section is not copied. However, during C run-time initialization—execution of the crt0 code block—some of the other code sections may be copied to volatile memory in preparation for running the application.

For more information about the behavior of the crt0 code, refer to "crt0 Initialization".

Altera recommends that you avoid this configuration during the normal development cycle because downloading the compiled application requires reprogramming the flash memory. In addition, software breakpoint capabilities require that hardware breakpoints be enabled for the Nios II processor when using this configuration.

Prepare for BSP configuration by following these steps to configure your application to boot and run from flash memory:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in flash memory. Configure the reset address and flash memory addresses in Qsys.
2. **Text section linker setting**—Ensure that the `.text` section maps to the flash memory address region. You can examine and modify section mappings in the Linker Script tab in the BSP Editor. Alternatively, use the following Tcl command:

   ```
   add_section_mapping .text ext_flash
   ```
3. **Other sections linker setting**—Ensure that all of the other sections, with the possible exception of the `.rodata` section, are mapped to volatile memory regions. The `.rodata` section can map to a flash-memory region.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in the table below.

Table 5-1: BSP Settings to Boot and Run from Flash Memory

BSP Setting Name	Value
`hal.linker.allow_code_at_reset`	1
`hal.linker.enable_alt_load`	1
`hal.linker.enable_alt_load_copy_rwdata`	1
`hal.linker.enable_alt_load_copy_exceptions`	1
`hal.linker.enable_alt_load_copy_rodata`	1

If your application contains custom memory sections, you must manually load the custom sections. Use the `alt_load_section()` HAL library function to ensure that these sections are loaded before your program runs.

The HAL BSP library disables the flash memory write capability to prevent accidental overwrite of the application image.

Related Information

- **crt0 Initialization** on page 4-42
- **AN736: Nios II Processor Booting Methods in MAX 10 FPGA Devices**

Booting From Flash Memory and Running From Volatile Memory

If your application image is stored in flash memory, but executes from volatile memory with assistance from a boot loader program, prepare for BSP configuration by following these steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is an address in flash memory. Configure this option using Qsys.
2. **Text section linker setting**—Ensure that the `.text` section maps to a volatile region of system memory, and not to the flash memory.
3. **Other sections linker setting**—Ensure that all of the other sections, with the possible exception of the `.rodata` section, are mapped to volatile memory regions. The `.rodata` section can map to a flash-memory region.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in the table below.

Table 5-2: BSP Settings to Boot from Flash Memory and Run from Volatile Memory

BSP Setting Name	Value
`hal.linker.allow_code_at_reset`	0
`hal.linker.enable_alt_load`	0
`hal.linker.enable_alt_load_copy_rwdata`	0
`hal.linker.enable_alt_load_copy_exceptions`	0
`hal.linker.enable_alt_load_copy_rodata`	0

Booting and Running From Volatile Memory

This configuration is use in cases where the Nios II processor's memory is loaded externally by another processor or interconnect switch fabric master port. In this case, prepare for BSP configuration by performing the same steps as in “Booting From Flash Memory and Running From Volatile Memory”,

except that the Nios II processor reset address should be changed to the memory that holds the code that the processor executes initially. Prepare for BSP configuration by following these steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in volatile memory. Configure this option using Qsys.
2. **Text section linker setting**—Ensure that the .text section maps to the reset address memory.
3. **Other sections linker setting**—Ensure that all of the other sections, including the .rodata section, also map to the reset address memory.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in the table below.

Table 5-3: BSP Settings to Boot and Run from Volatile Memory

BSP Setting Name	Value
`hal.linker.allow_code_at_reset`	1
`hal.linker.enable_alt_load`	0
`hal.linker.enable_alt_load_copy_rwdata`	0
`hal.linker.enable_alt_load_copy_exceptions`	0
`hal.linker.enable_alt_load_copy_rodata`	0

This type of boot loading and sequencing requires additional supporting hardware modifications, which are beyond the scope of this section.

Related Information

Booting From Flash Memory and Running From Volatile Memory on page 5-5

Booting From Altera EPCS Memory and Running From Volatile Memory

This configuration is a special case of the configuration described in "Booting From Flash Memory and Running From Volatile Memory". However, in this configuration, the processor does not perform the initial boot loading operation. The EPCS flash memory stores the FPGA hardware image and the application image. During system power up, the FPGA configures itself from EPCS memory. Then the Nios II processor resets control to a small FPGA memory resource in the EPCS memory controller, and executes a small boot loader application that copies the application from EPCS memory to the application's run-time location.

To make this configuration work, you must instantiate the EPCS device controller core in your system hardware. Add the component using Qsys.

Prepare for BSP configuration by following these steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in the EPCS memory controller. Configure this option using Qsys.
2. **Text section linker setting**—Ensure that the `.text` section maps to a volatile region of system memory.
3. **Other sections linker setting**—Ensure that all of the other sections, including the `.rodata` section, map to volatile memory.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in the table below.

Table 5-4: BSP Settings to Boot from EPCS and Run from Volatile Memory

BSP Setting Name	Value
hal.linker.allow_code_at_reset	0
hal.linker.enable_alt_load	0
hal.linker.enable_alt_load_copy_rwdata	0
hal.linker.enable_alt_load_copy_exceptions	0
hal.linker.enable_alt_load_copy_rodata	0

Related Information

- **EPCS to EPCQ Migration Guideline**
- **Booting From Flash Memory and Running From Volatile Memory** on page 5-5

Booting and Running From FPGA Memory

In this configuration, the program is loaded in and runs from internal FPGA memory resources. The FPGA memory resources are automatically configured when the FPGA device is configured, so no additional boot loading operations are required.

Prepare for BSP configuration by following these steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in the FPGA internal memory. Configure this option using Qsys.
2. **Text section linker setting**—Ensure that the `.text` section maps to the internal FPGA memory.
3. **Other sections linker setting**—Ensure that all of the other sections map to the internal FPGA memory.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in the table below.

Table 5-5: BSP Settings to Boot and Run from FPGA Memory

BSP Setting Name	Value
hal.linker.allow_code_at_reset	1
hal.linker.enable_alt_load	0
hal.linker.enable_alt_load_copy_rwdata	0
hal.linker.enable_alt_load_copy_exceptions	0
hal.linker.enable_alt_load_copy_rodata	0

This configuration requires that you generate FPGA memory Hexadecimal (Altera-format) Files (**.hex**) for compilation to the FPGA image. This step is described in the following section.

Generating and Programming System Memory Images

After you configure your linker settings and boot loader configuration and build the application image .elf file, you must create a memory programming file. The flow for creating the memory programming file depends on your choice of FPGA, flash, or EPCS memory.

The easiest way to generate the memory files for your system is to use the application-generated makefile targets.

The available mem_init.mk targets are listed in the "Common BSP Tasks" section in the Nios II Software Build Tools chapter of the *Nios II Gen2 Software Developer's Handbook*. You can also perform the same process manually, as shown in the following sections.

Generating memory programming files is not necessary if you want to download and run the application on the target system, for example, during the development and debug cycle.

Related Information

Nios II Software Build Tools

Programming FPGA Memory with the Nios II Software Build Tools Command Line

If your software application is designed to run from an internal FPGA memory resource, you must convert the application image .elf file to one or more **.hex** memory files. The Quartus Prime software compiles these .hex memory files to a **.sof** file. When this image is loaded in the FPGA it initializes the internal memory blocks.

To create a .hex memory file from your .elf file, type the following command:

```
elf2hex <myapp>.elf <start_addr> <end_addr> --width=<data_width> <hex_filename>.hex
```

This command creates a .hex memory file from application image <*myapp*>**.elf**, using data between <*start_addr*> and <*end_addr*>, formatted for memory of width <*data_width*>. The command places the output in the file <*hex_filename*>**.hex**. For information about **elf2hex** command-line arguments, type `elf2hex --help`.

Compile the **.hex** memory files to an FPGA image using the Quartus Prime software. Initializing FPGA memory resources requires some knowledge of Qsys and the Quartus Prime software.

Configuring and Programming Flash Memory in Nios II Software Build Tools for Eclipse

The Nios II Software Build Tools for Eclipse provide flash programmer utilities to help you manage and program the contents of flash memory.

The flash programmer allows you to program any combination of software, hardware, and binary data into flash memory in one operation.

"Configuring and Programming Flash Memory from the Command Line" describes several common tasks that you can perform in the Flash Programmer in command-line mode. Most of these tasks can also be performed with the Flash Programmer GUI in the Nios II Software Build Tools for Eclipse.

For information about using the Flash Programmer, refer to "Programming Flash in Altera Embedded Systems" in the Getting Started with the Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Getting Started with the Graphical User Interface**
- **Configuring and Programming Flash Memory from the Command Line** on page 5-9

Configuring and Programming Flash Memory from the Command Line

After you configure and build your BSP project and your application image **.elf** file, you must generate a flash programming file. The **nios2-flash-programmer** tool uses this file to configure the flash memory device through a programming cable, such as the USB-Blaster cable.

Creating a Flash Image File

If a boot loader application is required in your system, then you must first create a flash image file for your system. This section shows some standard commands in the Nios II Software Build Tools command line to create a flash image file. The section does not address the case of programming and configuring the FPGA image from flash memory.

The following standard commands create a flash image file for your flash memory device:

- **Boot loader required and EPCS flash device used**—To create an EPCS flash device image, type the following command:

```
elf2flash --epcs --after=<standard>.flash --input=<myapp>.elf \
--output=<myapp>.flash
```

  This command converts the application image in the file <*myapp*>**.elf** to a flash record format, and creates the new file <*myapp*>**.flash** that contains the new flash record appended to the FPGA hardware image in <*standard*>**.flash**.
- **Boot loader required and CFI flash memory used**—To create a CFI flash memory image, type the following command:

```
elf2flash --base=0x0 --reset=0x0 --end=0x1000000 \
--boot=<boot_loader_cfi>.srec \
--input=<myapp>.elf --output=<myapp>.flash
```

- This command converts the application image in the file <*myapp*>**.elf** to a flash record format, and creates the new file <*myapp*>**.flash** that contains the new flash record appended to the CFI boot loader in <*boot_loader_cfi*>**.srec**. The flash record is to be downloaded to the reset address of the Nios II processor, `0x0`, and the base address of the flash device is `0x0`. If you use the Altera-supplied boot loader, your user-created program sections are also loaded from the flash memory to their run-time locations.
- **No boot loader required and CFI flash memory used**—To create a CFI flash memory image, if no boot loader is required, type the following command:

```
elf2flash --base=0x0 --reset=0x0 --end=0x1000000 \
--input=<myapp>.elf --output=<myapp>.flash
```

  This command and its effect are almost identical to those of the command to create a CFI flash memory image if a boot loader is required. In this case, no boot loader is required, and therefore the `--boot` command-line option is not present.

The Nios II EDS includes two precompiled boot loaders for your use, one for CFI flash devices and another for EPCS flash devices. The source code for these boot loaders can be found in the `<Nios II EDS install dir>/components/altera_nios2/boot_loader_sources/ directory`.

Document Revision History

Table 5-6: Nios II Configuration and Booting Solutions Revision History

Date	Version	Changes
December 2016	2016.12.19	Initial release.

Nios II Debug, Verification, and Simulation

6

2016.12.19

ED_HANDBOOK Subscribe Send Feedback

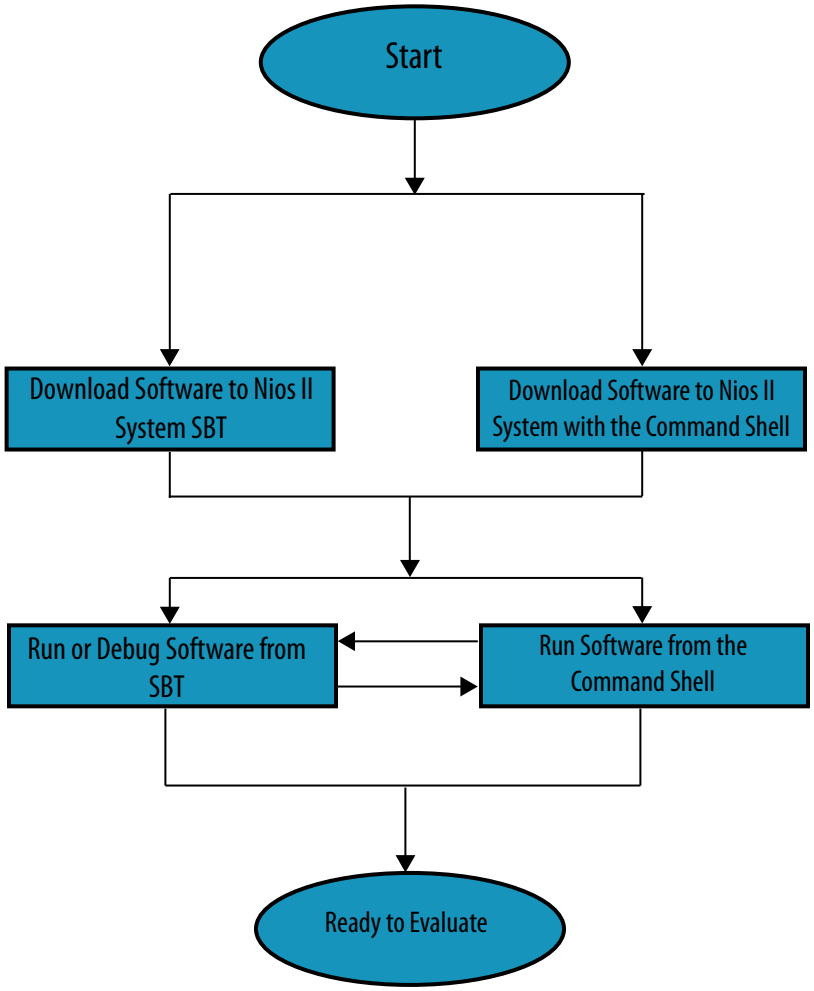
This chapter introduces the best practices for debugging the Nios II processor and verification for embedded design and simulation. Debugging and verifying an embedded system involves hardware and software components. To successfully debug an embedded system requires expertise in both hardware and software. This chapter helps you understand several tools and techniques that are useful in debugging, verifying, and bring up the embedded system.

Software Debugging Options

The Nios II EDS provides the following programs to aid in debugging your hardware and software system:

- The Nios II SBT for Eclipse debugger
- Several distinct interfaces to the GNU Debugger (GDB)
- A Nios II-specific implementation of the First Silicon Solutions, Inc.
- System Console, a system debug console

Figure 6-1: Nios II Software Development Flows: Testing Software



You can begin debugging software immediately using the built-in Nios II SBT for Eclipse debugger. This debugging environment includes advanced features such as trace, watchpoints, and hardware breakpoints.

The Nios II EDS includes the following three interfaces to the GDB debugger:

- GDB console (accessible through the Nios II SBT for Eclipse)
- Standard GDB client (nios2-elf-gdb)
- Insight GDB interface (Tcl/Tk based GUI)

Additional GDB interfaces such as Data Display Debugger (DDD), and Curses GDB (CGDB) interface also function with the Nios II version of the GDB debugger.

For more information about these interfaces to the GDB debugger, refer to the "Nios II Command-Line Tools" and "Debugging Nios II Designs" chapters of the Embedded Design Handbook.

The System Console is a system debug console that provides the Qsys designer with a Tcl-based, scriptable command-line interface for performing system or individual component testing.

For detailed information about the System Console, refer to the "Analyzing and Debugging Designs with the System Console" chapter in volume 3 of the Quartus Prime Handbook. Online training is available at the Altera Training page of the Altera website.

Third party debugging environments are also available from vendors such as Lauterbach Datentechnik GmbH and First Silicon Solutions, Inc.

Related Information

- **Debugging Nios II Designs** on page 6-3
- **Nios II Command-Line Tools** on page 4-1
- **Verification and Board Bring-Up** on page 6-18
- **Analyzing and Debugging Designs with System Console**
- **Altera Training**

Debugging Nios II Designs

This section describes best practices for debugging Nios II processor software designs. Debugging these designs involves debugging both hardware and software, which requires familiarity with multiple disciplines. Successful debugging requires expertise in board layout, FPGA configuration, and Nios II software tools and application software. This section includes the following topics that discuss debugging techniques and tools to address difficult embedded design problems:

Related Information

- **Debuggers** on page 6-3
- **Run-Time Analysis Debug Techniques** on page 6-13

Debuggers

The Nios II development environments offer several tools for debugging Nios II software systems. This section describes the debugging capabilities available in the following development environments:

Related Information

- **Nios II Software Development Tools** on page 6-3
- **SignalTap II Embedded Logic Analyzer** on page 6-12
- **Lauterbach Trace32 Debugger and PowerTrace Hardware** on page 6-12
- **Data Display Debuggers** on page 6-13

Nios II Software Development Tools

The Nios II Software Build Tools for Eclipse is a graphical user interface (GUI) that supports creating, modifying, building, running, and debugging Nios II programs. The Nios II Software Build Tools for the command line are command-line utilities available from a Nios II Command Shell. The Nios II Software Build Tools for Eclipse use the same underlying utilities and scripts as the Nios II Software Build Tools for the command line. Using the Software Build Tools provides fine control over the build process and project settings.

Qsys is a system development tool for creating systems including processors, peripherals, and memories. The tool enables you to define and generate a complete FPGA system very efficiently. Qsys does not require that your system contain a Nios II processor. However, it provides complete support for integrating Nios II processors in your system, including some critical debugging features.

The following sections describe debugging tools and support features available in the Nios II software development tools:

Related Information

- **Nios II System ID** on page 6-4
- **Project Templates** on page 6-6
- **Configuration Options** on page 6-6
- **Nios II GDB Console and GDB Commands** on page 6-8
- **Nios II Console View and stdio Library Functions** on page 6-9
- **Importing Projects Created Using the Nios II Software Build Tools** on page 6-10
- **Selecting a Processor Instance in a Multiple Processor Design** on page 6-10

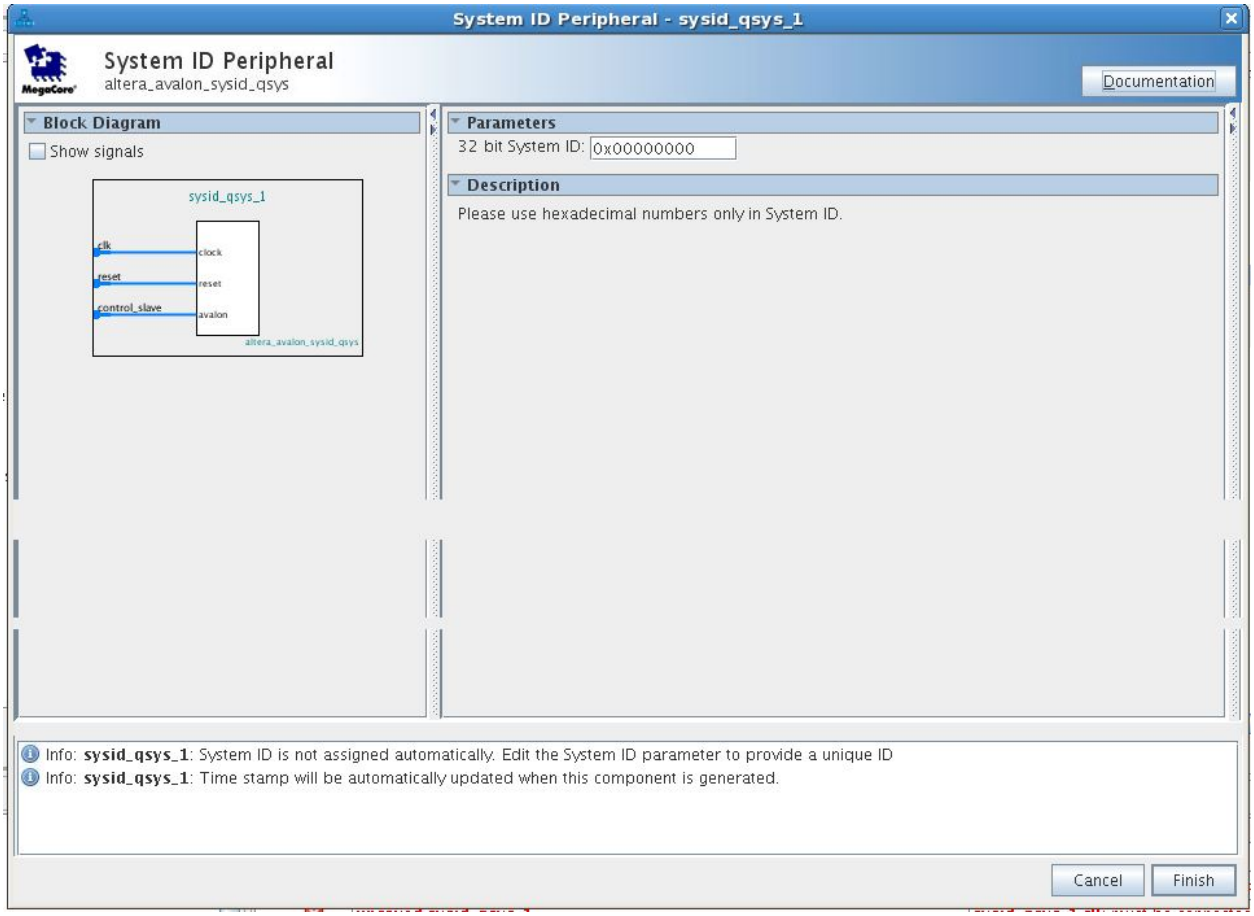
Nios II System ID

The system identifier (ID) feature is available as a system component in Qsys. The component allows the debugger to identify attempts to download software projects with BSP projects that were generated for a different Qsys system. This feature protects you from inadvertently using an Executable and Linking Format (.elf) file built for a Nios II hardware design that is not currently loaded in the FPGA. If your application image does not run on the hardware implementation for which it was compiled, the results are unpredictable.

To start your design with this basic safety net, in the Nios II Software Build Tools for Eclipse **Debug Configurations** dialog box, in the **Target Connection** tab, ensure that **Ignore mismatched system ID** is not turned on.

The system ID feature requires that the Qsys design include a system ID component. In the figure below shows an Qsys system with a system ID component.

Figure 6-2: Qsys System With System ID Component



For more information about the System ID component, refer to the System ID Core chapter in the *Embedded Peripheral IP User Guide*.

Related Information

System ID Core

Project Templates

The Nios II Software Build Tools for Eclipse help you to create a simple, small, and pretested software project to test a new board.

The Nios II Software Build Tools for Eclipse provide a mechanism to create new software projects using project templates. To create a simple test program to test a new board, perform the following steps:

1. In the Nios II perspective, on the File menu, point to **New**, and click **Nios II Application and BSP from Template**.

   The New Project wizard for Nios II C/C++ application projects appears.
2. Specify the Qsys information (**.sopcinfo**) file for your design. The folder in which this file is located is your project directory.
3. If your hardware design contains multiple Nios II processors, in the **CPU** list, click the processor you wish to run this application software.
4. Specify a project name.
5. In the **Templates** list, click **Hello World Small**.
6. Click **Next**.
7. Click **Finish**.

The Hello World Small template is a very simple, small application. Using a simple, small application minimizes the number of potential failures that can occur as you bring up a new piece of hardware.

To create a new project for which you already have source code, perform the preceding steps with the following exceptions:

- In step 5, click **Blank Project**.
- After you perform step 7, perform the following steps:

  1. Create the new directory <your_project_directory>/**software**/<project_name>/**source**, where <project_name> is the project name you specified in step 4.
  2. Copy your source code files to the new project by copying them to the new <your_project_directory>/**software**/<project_name>/**source** directory.
  3. In the **Project Explorer** tab, right-click your application project name, and click **Refresh**. The new **source** folder appears under your application project name.

Configuration Options

The following Nios II Software Build Tools for Eclipse configuration options increase the amount of debugging information available for your application image **.elf** file:

Related Information

- **Objdump File** on page 6-6
- **Show Make Commands** on page 6-8
- **Show Line Numbers** on page 6-8

Objdump File

You can direct the Nios II build process to generate helpful information about your .elf file in an object dump text file (**.objdump**). The **.objdump** file contains information about the memory sections and their layout, the addresses of functions, and the original C source code interleaved with the assembly code. The example below shows part of the C and assembly code section of an **.objdump** file for the Nios II built-in Hello World Small project.

Example 6-1: Piece of Code in .objdump File From Hello World Small Project

```
06000170 <main>:

include "sys/alt_stdio.h"

int main()
{
6000170:deffff04 addisp,sp,-4
alt_putstr("Hello from Nios II!\n");
6000174:01018034 movhir4,1536
6000178:2102ba04 addir4,r4,2792
600017c:dfc00015 stwra,0(sp)
6000180:60001c00 call60001c0 <alt_putstr>
6000184:003fff06 br6000184 <main+0x14>

06000188 <alt_main>:
* the users application, i.e. main().
*/

void alt_main (void)
{
6000188:deffff04 addisp,sp,-4
600018c:dfc00015 stwra,0(sp)

static ALT_INLINE void ALT_ALWAYS_INLINE
alt_irq_init (const void* base)
{
NIOS2_WRITE_IENABLE (0);
6000190:000170fa wrctlienable,zero
NIOS2_WRITE_STATUS (NIOS2_STATUS_PIE_MSK);
6000194:00800044 movir2,1
6000198:1001703a wrctlstatus,r2
```

To enable this option in the Nios II Software Build Tools for Eclipse, perform the following steps:

1. In the Project Explorer window, right-click your application project and click **Properties**.
2. On the list to the left, click **Nios II Application Properties**.
3. On the **Nios II Application Properties** page, turn on **Create object dump**.
4. Click **Apply**.
5. Click **OK**.

After the next build, the **.objdump** file is found in the same directory as the newly built **.elf** file.

After the next build generates the .elf file, the build runs the nios2-elf-objdump command with the options `--disassemble-all`, `--source`, and `--all-headers` on the generated .elf file.

In the Nios II user-managed tool flow, you can edit the settings in the application makefile that determine the options with which the **nios2-elf-objdump** command runs. Running the **create-this-app** script, or the **nios2-app-generate-makefile** script, creates the following lines in the application makefile:

```
#Options to control objdump.
CREATE_OBJDUMP := 1
OBJDUMP_INCLUDE_SOURCE :=0
OBJDUMP_FULL_CONTENTS := 0
```

Edit these options to control the **.objdump** file according to your preferences for the project:

- `CREATE_OBJDUMP`—The value 1 directs **nios2-elf-objdump** to run with the options `--disassemble`, `--syms`, `--all-header`, and `--source`.
- `OBJDUMP_INCLUDE_SOURCE`—The value 1 adds the option `--source` to the **nios2-elf-objdump** command line.
- `OBJDUMP_FULL_CONTENTS`—The value 1 adds the option `--full-contents` to the **nios2-elf-objdump** command line.

For detailed information about the information each command-line option generates, in a Nios II Command Shell, type the following command:

```
nios2-elf-objdump --help
```

Show Make Commands

To enable a verbose mode for the make command, in which the individual Makefile commands appear in the display as they are run, in the Nios II Software Build Tools for Eclipse, perform the following steps:

1. In the Project Explorer window, right-click your application project and click **Properties**.
2. On the list to the left, click **C/C++ Build**.
3. On the **C/C++ Build** page, turn off **Use default build command**.
4. For **Build command**, type `make -d`.
5. Click **Apply**.
6. Click **OK**.

Show Line Numbers

To enable display of C source-code line numbers in the Nios II Software Build Tools for Eclipse, follow these steps:

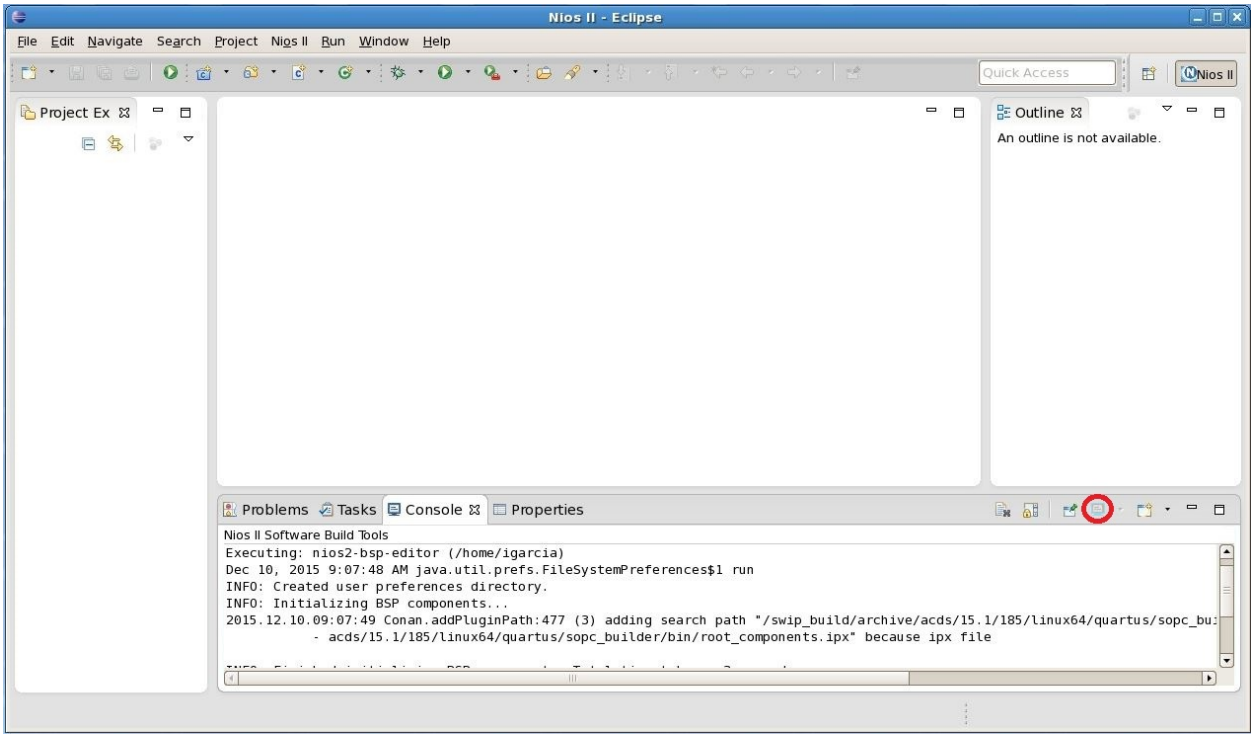
1. On the Window menu, click **Preferences**.
2. On the list to the left, under **General**, under **Editors**, select **Text Editors**.
3. On the **Text Editors** page, turn on **Show line numbers**.
4. Click **Apply**.
5. Click **OK**.

Nios II GDB Console and GDB Commands

The Nios II GNU Debugger (GDB) console allows you to send GDB commands to the Nios II processor directly.

To display this console, which allows you to enter your own GDB commands, click the blue monitor icon on the lower right corner of the Nios II Debug perspective. (If the Nios II Debug perspective is not displayed, on the Window menu, click **Open Perspective**, and click **Other** to view the available perspectives). If multiple consoles are connected, click the black arrow next to the blue monitor icon to list the available consoles. On the list, select the GDB console. The figure below shows the console list icon— the blue monitor icon and black arrow—that allow you to display the GDB console.

Figure 6-3: Console List Icon



An example of a useful command you can enter in the GDB console is:

```
dump binary memory <file> <start_addr> <end_addr>
```

This command dumps the contents of a specified address range in memory to a file on the host computer. The file type is binary. You can view the generated binary file using the HexEdit hexadecimal-format editor that is available from the HexEdit website.

Related Information

www.expertcomsoft.com

Nios II Console View and stdio Library Functions

When debugging I/O behavior, you should be aware of whether your Nios II software application outputs characters using the `printf()` function from the stdio library or the `alt_log_printf()` function. The two functions behave slightly differently, resulting in different system and I/O blocking behavior.

The `alt_log_printf()` function bypasses HAL device drivers and writes directly to the component device registers. The behavior of the two functions may also differ depending on whether you enable the reduced-driver option, whether you set your nios2-terminal session or the Nios II Console view in the Nios II Software Build Tools for Eclipse to use a UART or a jtag_uart as the standard output device, and whether the `O_NONBLOCK` control code is set. In general, enabling the reduced-driver option disables interrupts, which can affect blocking in jtag_uart devices.

To enable the reduced-drivers option, perform the following steps:

1. In the Nios II Software build Tools for Eclipse, in the Project Explorer window, right-click your BSP project.
2. Point to **Nios II** and click **BSP Editor**. The BSP Editor appears.
3. In the BSP Editor, in the **Settings** tab, under **Common**, under **hal**, click **enable_reduced_device_drivers**.
4. Click **Generate**.

For more information about the `alt_log_printf()` function, refer to "Using Character-Mode Devices" in the Developing Programs Using the Hardware Abstraction Layer chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Developing Programs Using the Hardware Abstraction Layer

Importing Projects Created Using the Nios II Software Build Tools

Whether a project is created and built using the Nios II Software Build Tools for Eclipse or using the Nios II Software Build Tools command line, you can debug the resulting .elf image file in the Nios II Software Build Tools for Eclipse.

For information about how to import a project created with the Nios II Software Build Tools command line to the Nios II Software Build Tools for Eclipse, refer to "Importing a Command-Line Project" in the Getting Started with the Graphical User Interface chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

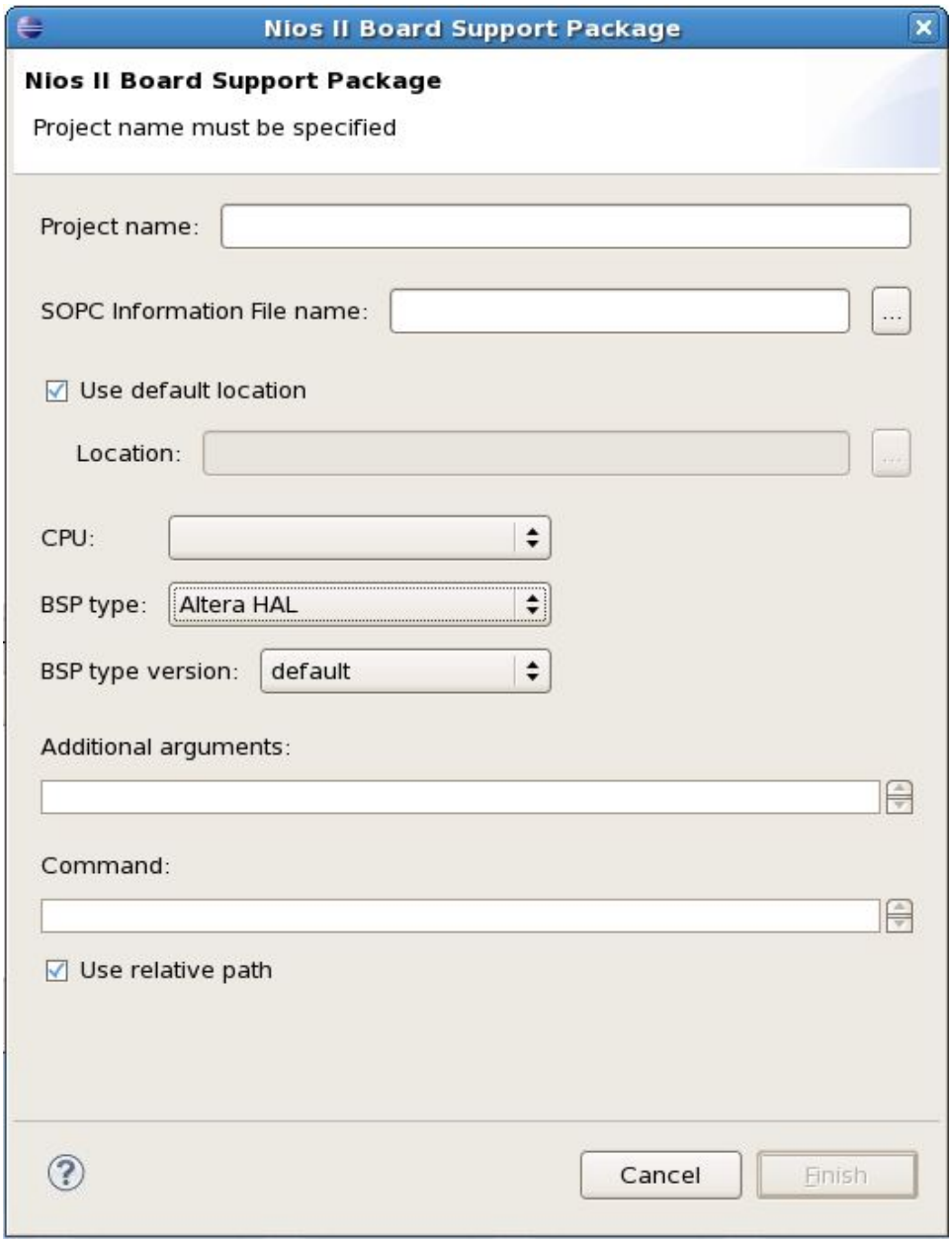
Getting Started with the Graphical User Interface

Selecting a Processor Instance in a Multiple Processor Design

In a design with multiple Nios II processors, you must create a different software project for each processor. When you create an application project, the Nios II Software Build Tools for Eclipse require that you specify a Board Support Package (BSP) project. If a BSP for your application project does not yet exist, you can create one. For BSP generation, you must specify the CPU to which the application project is targeted.

The figure below shows how you specify the CPU for the BSP in the Nios II Software Build Tools for Eclipse. The **Nios II Board Support Package** page of the New Project wizard collects the information required for BSP creation. This page derives the list of available CPU choices from the **.sopcinfo** file for the system.

Figure 6-4: Nios II Software Build Tools for Eclipse Board Support Package Page— CPU Selection



From the Nios II Command Shell, the **jtagconfig –n** command identifies available JTAG devices and the number of CPUs in the subsystem connected to each JTAG device. The example below shows the system response to a **jtagconfig -n** command.

Example 6-2: Two-FPGA System Response to jtagconfig Command

```
[Qsys]$ jtagconfig -n
1) USB-Blaster [USB-0]
120930DD EP2S60
Node 19104600
Node 0C006E00
2) USB-Blaster [USB-1]
020B40DD EP2C35
Node 19104601
Node 19104602
Node 19104600
Node 0C006E00
```

The response in the example lists two different FPGAs, connected to the running JTAG server through different USB-Blaster cables. The cable attached to the USB-0 port is connected to a JTAG node in a Qsys subsystem with a single Nios II core. The cable attached to the USB-1 port is connected to a JTAG node in a Qsys subsystem with three Nios II cores. The node numbers represent JTAG nodes inside the FPGA. The appearance of the node number `0x191046xx` in the response confirms that your FPGA implementation has a Nios II processor with a JTAG debug module. The appearance of a node number `0x0C006Exx` in the response confirms that the FPGA implementation has a JTAG UART component. The CPU instances are identified by the least significant byte of the nodes beginning with 191. The JTAG UART instances are identified by the least significant byte of the nodes beginning with 0C. Instance IDs begin with 0.

Only the CPUs that have JTAG debug modules appear in the listing. Use this listing to confirm you have created JTAG debug modules for the Nios II processors you intended.

SignalTap II Embedded Logic Analyzer

The SignalTap II embedded logic analyzer can help you to catch some software-related problems, such as an interrupt service routine that does not properly clear the interrupt signal.

For information about the SignalTap II embedded logic analyzer, refer to the Design Debugging Using the SignalTap II Embedded Logic Analyzer chapter in volume 3 of the *Quartus Prime Handbook* and the Verification and Board Bring-Up chapter of this handbook.

The Nios II plug-in for the SignalTap II embedded logic analyzer enables you to capture a Nios II processor's program execution.

For more information about the Nios II plug-in for the SignalTap II embedded logic analyzer, refer to *AN446: Debugging Nios II Systems with the SignalTap II Logic Analyzer*.

Related Information

- **Verification and Board Bring-Up** on page 6-18
- **Design Debugging Using the SignalTap II Logic Analyzer**
- **AN446: Debugging Nios II Systems with the SignalTap II Logic Analyzer.**

Lauterbach Trace32 Debugger and PowerTrace Hardware

Lauterbach Datentechnik GmbH (Lauterbach) provides the Trace32 ICD-Debugger for the Nios II processor. The product contains both hardware and software. In addition to a connection for the 10-pin JTAG connector that is used for the Altera USB-Blaster cable, the PowerTrace hardware has a 38-pin mictor connection option.

Lauterbach also provides a module for off-chip trace capture and an instruction-set simulator for Nios II systems.

The order in which devices are powered up is critical. The Lauterbach PowerTrace hardware must always be powered when power to the FPGA hardware is applied or terminated. The Lauterbach PowerTrace hardware's protection circuitry is enabled after the module is powered up.

For more information about the Lauterbach PowerTrace hardware and the required power-up sequence, refer to *AN543: Debugging Nios II Software Using the Lauterbach Debugger*.

Related Information

- **AN543: Debugging Nios II Software Using the Lauterbach Debugger**
- **www.lauterbach.com**

Data Display Debuggers

Another alternative debugger is the Data Display Debugger (DDD). This debugger is compatible with GDB commands—it is a user interface to the GDB debugger—and can therefore be used to debug Nios II software designs. The DDD can display data structures as graphs.

Run-Time Analysis Debug Techniques

This section discusses methods and tools available to analyze a running software system.

Software Profiling

Altera provides the following tools to profile the run-time behavior of your software system:

- **GNU profiler**—The Nios II EDS toolchain includes the gprof utility for profiling your application. This method of profiling reports how long various functions run in your application.
- **High resolution timer**—The Qsys timer peripheral is a simple time counter that can determine the amount of time a given subroutine or code segment runs. You can read it at various points in the source code to calculate elapsed time between timer samples.
- **Performance counter peripheral**—The Qsys performance counter peripheral can profile several different sections of code with a series of counter peripherals. This peripheral includes a simple software API that enables you to print out the results of these timers through the Nios II processor's stdio services.

For more information about how to profile your software application, refer to *AN391: Profiling Nios II Systems*.

For additional information about the Qsys timer peripheral, refer to the Interval Timer Core chapter in the *Embedded Peripherals IP User Guide* and to the Developing Nios II Software chapter of this handbook.

For additional information about the Qsys performance counter peripheral, refer to the Performance Counter Core chapter in the *Embedded Peripherals IP User Guide*.

Related Information

- **Developing Nios II Software** on page 4-18
- **AN391: Profiling Nios II Systems**
- **Interval Timer Core**
- **Performance Counter Core**

Watchpoints

Watchpoints provide a powerful method to capture all writes to a global variable that appears to be corrupted. The Nios II Software Build Tools for Eclipse support watchpoints directly.

For more information about watchpoints, refer to the Nios II online Help. In Nios II Software Build Tools for Eclipse, on the Help menu, click **Search**. In the search field, type watchpoint, and select the topic **Adding watchpoints**.

To enable watchpoints, you must configure the Nios II processor's debug level in Qsys to debug level 2 or higher. To configure the Nios II processor's debug level in Qsys to the appropriate level, perform the following steps:

1. On the Qsys System Contents tab, right-click the desired Nios II processor component. A list of options appears.
2. On the list, click Edit. The Nios II processor configuration page appears.
3. Click the JTAG Debug Module tab, shown in **Figure 6-5**
4. Select Level 2, Level 3, or Level 4.
5. Click Finish.

Depending on the debug level you select, a maximum of four watchpoints, or data triggers, are available. **Figure 6-5** shows the number of data triggers available for each debug level. The higher your debug level, the more logic resources you use on the FPGA.

For more information about the Nios II processor debug levels, refer to the Instantiating the Nios II Processor chapter in the *Nios II Gen2 Processor Reference Handbook*.

Related Information

Instantiating the Nios II Gen2 Processor

Stack Overflow

Stack overflow is a common problem in embedded systems, because their limited memory requires that your application have a limited stack size. When your system runs a real-time operating system, each running task has its own stack, increasing the probability of a stack overflow condition. As an example of how this condition may occur, consider a recursive function, such as a function that calculates a factorial value. In a typical implementation of this function, `factorial(n)` is the result of multiplying the value n by another invocation of the factorial function, `factorial(n- 1)`. For large values of n, this recursive function causes many call stack frames to be stored on the stack, until it eventually overflows before calculating the final function return value.

Enabling Run Time Stack Checking

Using the Nios II Software Build Tools for Eclipse, you can enable the HAL to check for stack overflow.

If you enable stack overflow checking and you register an instruction-related exception handler, on stack overflow, the HAL calls the instruction-related exception handler. If you enable stack overflow checking and do not register an instruction-related exception handler, and you enable a JTAG debug module for your Nios II processor, on stack overflow, execution pauses in the debugger, exactly as it does when the debugger encounters a breakpoint. To enable stack overflow checking, in the BSP Editor, in the **Settings** tab, under **Advanced**, under **hal**, click **enable_runtime_stack_checking**.

For information about the instruction-related exception handler, refer to "The Instruction-Related Exception Handler" in the Exception Handling chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Exception Handling**
- **Run Time Stack Checking And Exception Debugging**

Hardware Abstraction Layer (HAL)

The Altera HAL provides the interfaces and resources required by the device drivers for most Qsys system peripherals. You can customize and debug these drivers for your own Qsys system.

To learn more about debugging HAL device drivers and Qsys peripherals, refer to *AN459: Guidelines for Developing a Nios II HAL Device Driver*.

Related Information

AN459: Guidelines for Developing a Nios II HAL Device Driver

Breakpoints

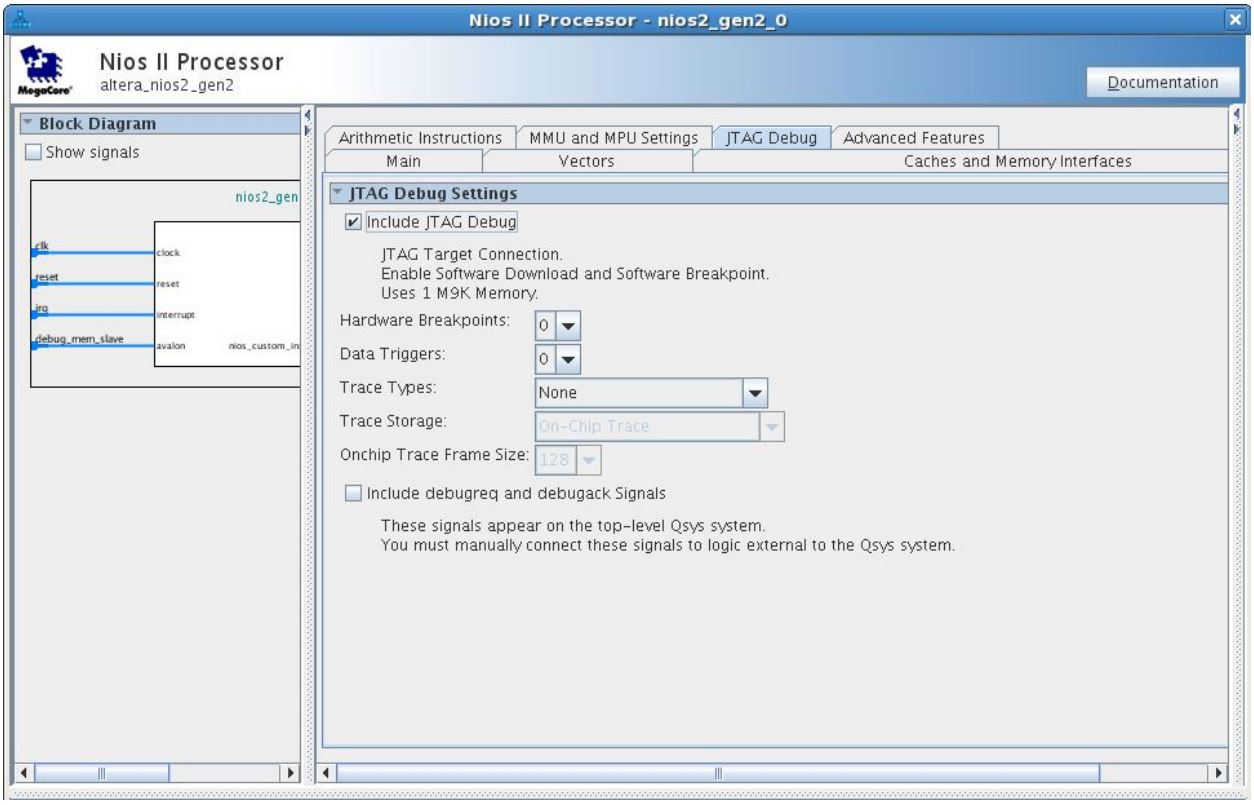
You can set hardware breakpoints on code located in read-only memory such as flash memory. If you set a breakpoint in a read-only area of memory, a hardware breakpoint, rather than a software breakpoint, is selected automatically.

To enable hardware breakpoints, you must configure the Nios II processor's debug level in Qsys to debug level 2 or higher. To configure the Nios II processor's debug level in Qsys to the appropriate level, perform the following steps:

1. On the Qsys **System Contents** tab, right-click the desired Nios II processor component.
2. On the right button pop-up menu, click **Edit**. The Nios II processor configuration page appears.
3. Click the **JTAG Debug Module** tab, shown in the figure below.
4. Select **Level 2**, **Level 3**, or **Level 4**.
5. Click **Finish**.

Depending on the debug level you select, a maximum of four hardware breakpoints are available. The figure below shows the number of hardware breakpoints available for each debug level. The higher your debug level, the more logic resources you use on the FPGA.

Figure 6-5: Nios II Processor — JTAG Debug Module — Qsys Configuration Page



For more information about the Nios II processor debug levels, refer to the Instantiating the Nios II Processor chapter in the *Nios II Gen2 Processor Reference Handbook*.

Related Information

Instantiating the Nios II Gen2 Processor

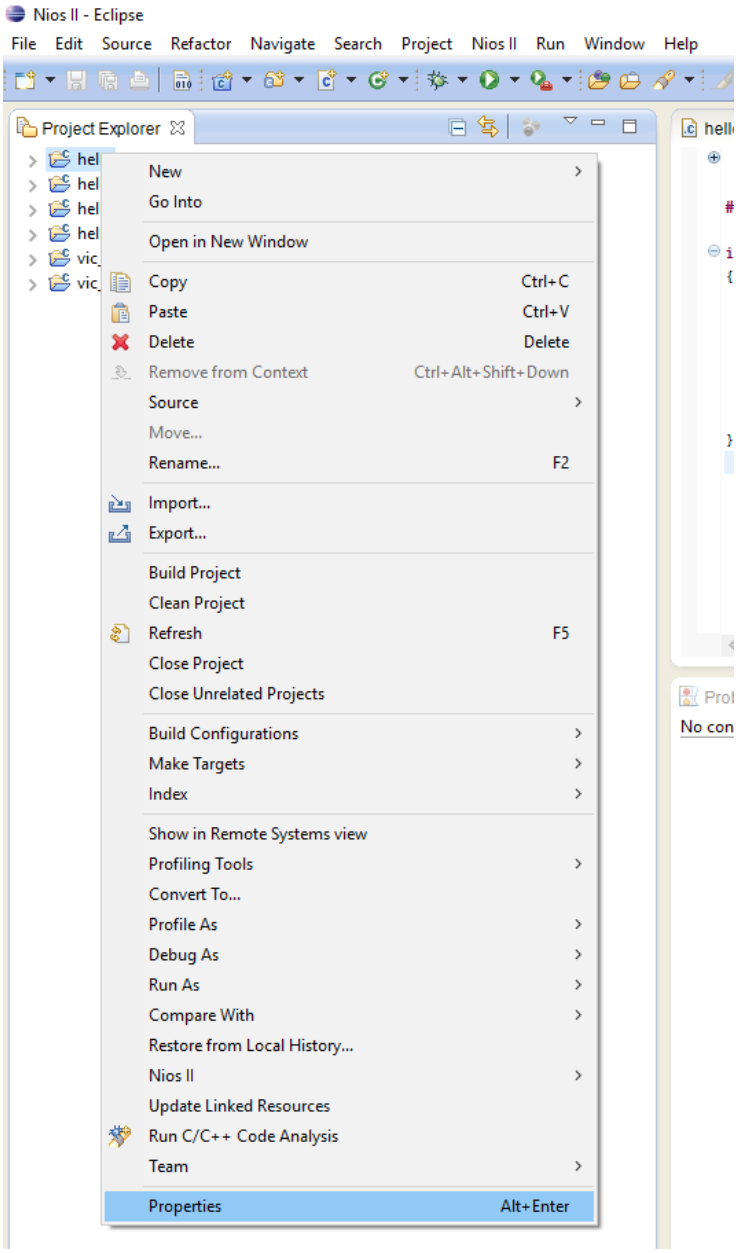
Debugger Stepping and Using No Optimizations

Use the None (**–O0**) optimization level compiler switch to disable optimizations for debugging. Otherwise, the breakpoint and stepping behavior of your debugger may not match the source code you wrote. This behavior mismatch between code execution and high-level original source code may occur even when you click the i button to use the instruction stepping mode at the assembler instruction level. This mismatch occurs because optimization and in-lining by the compiler eliminated some of your original source code.

To set the None (**–O0**) optimization level compiler switch in the Nios II Software Build Tools for Eclipse, perform the following steps:

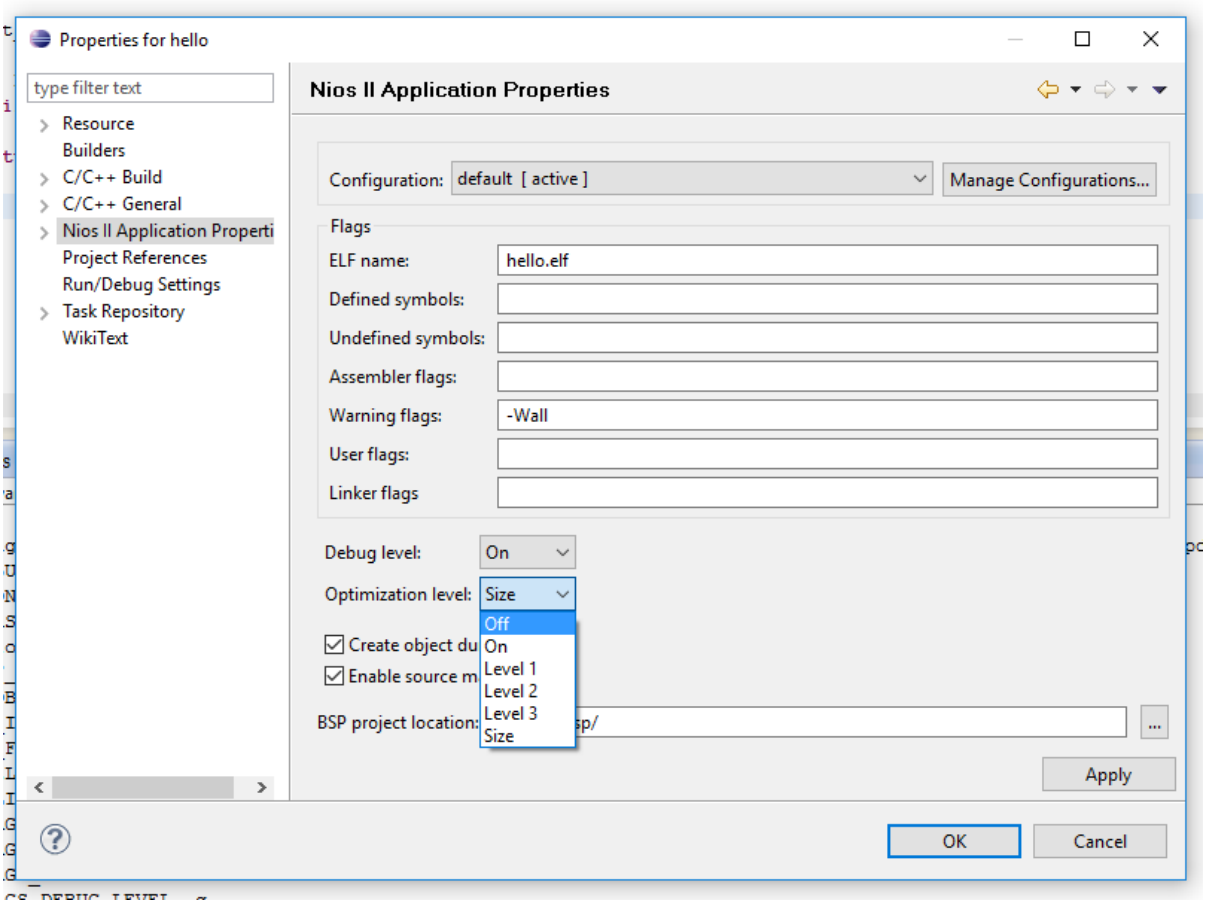
1. In the Nios II perspective, right-click your application project. A list of options appears.

Figure 6-6: Application Project Options



2. On the list, click **Properties**. The **Properties for** <*project name*> dialog box appears.
3. In the left pane, click **Nios II Application Properties**.

Figure 6-7: Nios Application Properties



4. In the **Optimization Level** list, select **Off**.
5. Click **Apply**.
6. Click **OK**

Verification and Board Bring-Up

This section provides an overview of the tools available in the Quartus Prime software and the Nios II Embedded Design Suite (EDS) that you can use to verify and bring up your embedded system.

This section covers the following topics:

- Verification Methods
- Board Bring-up
- System Verification

Verification Methods

Embedded systems can be difficult to debug because they have limited memory and I/O and consist of a mixture of hardware and software components. Altera provides the following tools and strategies to help you overcome these difficulties:

- System Console
- SignalTap II Embedded Logic Analyzer
- External Instrumentation
- Stimuli Generation

Prerequisites

To make effective use of this section, you should be familiar with the following topics:

- Defining and generating Nios II hardware systems with Qsys
- Compiling Nios II hardware systems with the Quartus Prime development software

System Console

You can use the System Console to perform low–level debugging of a Qsys system. You access the System Console functionality in command line mode. You can work interactively or run a Tcl script. The System Console prints responses to your commands in the terminal window. To facilitate debugging with the System Console, you can include one of the four Qsys components with interfaces that the System Console can use to send commands and receive data.

Table 6-1: Qsys Components for Communication with the System Console

Component Name	Debugs Components with the Following Interface Types
Nios II processor with JTAG debug enabled	Components that include an Avalon-MM slave interface. The JTAG debug module can also control the Nios II processor for debug functionality, including starting, stopping, and stepping the processor.
JTAG to Avalon master bridge	Components that include an Avalon-MM slave interface
Avalon Streaming (Avalon-ST) JTAG Interface	Components that include an Avalon-ST interface
JTAG UART	The JTAG UART is an Avalon-MM slave device that can be used in conjunction with the System Console to send and receive byte streams.

The System Console can also send and receive byte streams from any SLD node, whether it is instantiated in a Qsys component provided by Altera, a custom component, or part of your Quartus Prime project. However, this approach requires detailed knowledge of the JTAG commands.

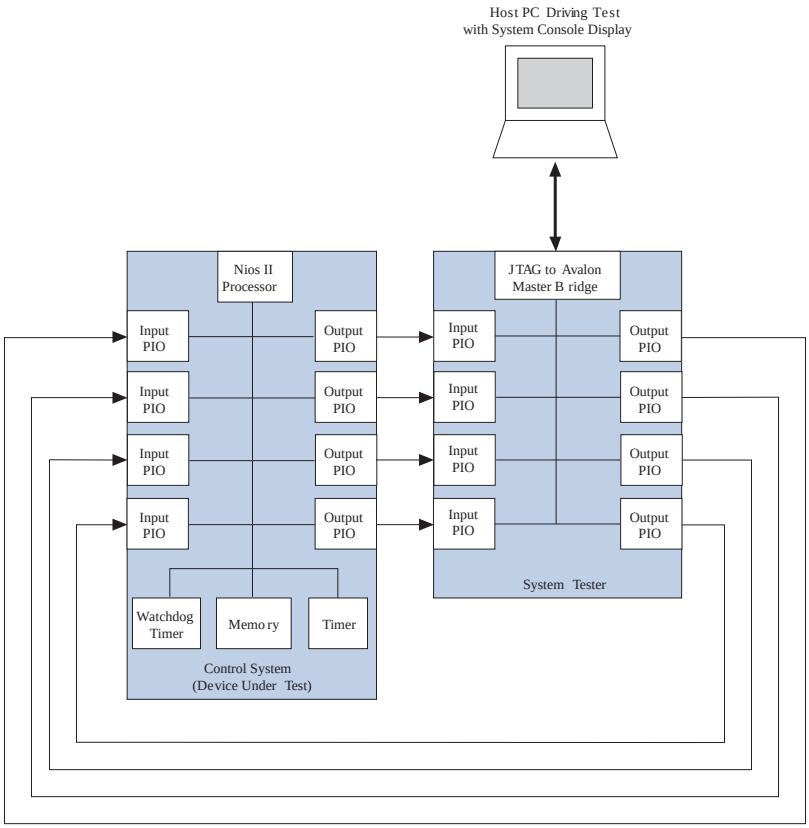
The System Console allows you to perform any of the following tasks:

- Access memory and peripherals
- Start or stop a Nios II processor
- Access a Nios II processor register set and step through software
- Verify JTAG connectivity
- Access the reset signal
- Sample the system clock

Using the System Console you can test your own custom components in real hardware without creating a testbench or writing test code for the Nios II processor. By coding a Tcl script to access a component with an Avalon-MM slave port, you create a testbench that abstracts the Avalon-MM master accesses to a higher level. You can use this strategy to quickly test components, I/O, or entire memory-mapped systems.

Embedded control systems typically include inputs such as sensors, outputs such as actuators, and a processor that determines the outputs based on input values. You can test your embedded control system in isolation by creating an additional system to exercise the embedded system in hardware. This approach allows you to perform automated testing of hardware-in-the-loop (HIL) by using the System Console to drive the inputs into the system and measure the outputs. This approach has the advantage of allowing you to test your embedded system without modifying the design. The figure below illustrates HIL testing using the System Console.

Figure 6-8: Hardware-in-the-Loop Testing Using the System Console



To learn more about the System Console refer to the System Console chapter of the *Quartus Prime Handbook Volume 3: Verification*.

Related Information

System Console

SignalTap II Embedded Logic Analyzer

The SignalTap II embedded logic analyzer is available in the Quartus Prime software. It reuses the JTAG pins of the FPGA and has a low Quartus Prime fitter priority, allowing it to be non-intrusive. Because this logic analyzer is integrated in your design automatically, it takes synchronized measurements without the

undesirable side effects of output pin capacitance or I/O delay. The SignalTap II embedded logic analyzer also supports Tcl scripting so that you can automate data capture, duplicating the functionality that external logic analyzers provide.

This logic analyzer can operate while other JTAG components, including the Nios II JTAG debug module and JTAG UART, are in use, allowing you to perform co-verification. You can use the plug-in support available with the SignalTap II embedded logic analyzer to enhance your debug capability with any of the following:

- Instruction address triggering
- Non-processor related triggering
- Software disassembly
- Instruction display (in hexadecimal or symbolic format)

You can also use this logic analyzer to capture data from your embedded system for analysis by the MATLAB software from Mathworks. The MATLAB software receives the data using the JTAG connection and can perform post processing analysis. Using looping structures, you can perform multiple data capture cycles automatically in the MATLAB software, instead of manually controlling the logic analyzer using the Quartus Prime design software.

Because the SignalTap II embedded logic analyzer uses the FPGA's JTAG connection, continuous data triggering may result in lost samples. For example, if you capture data continuously at 100 MHz, you should not expect all of your samples to be displayed in the logic analyzer GUI. The logic analyzer buffers the data at 100 MHz; however, if the JTAG interface becomes saturated, samples are lost.

To learn more about SignalTap II embedded logic analyzer and co-verification, refer to *AN446: Debugging Nios II Systems with the SignalTap II Embedded Logic Analyzer* and the *Quartus Prime Handbook Volume 3: Verification*.

Related Information

- **AN446: Debugging Nios II Systems with the SignalTap II Embedded Logic Analyzer**
- **Quartus Prime Handbook Volume 3: Verification**

External Instrumentation

If your design does not have enough on-chip memory to store trace buffers, you can use an external logic analyzer for debugging. External instrumentation is also necessary if you require any of the following:

- Data collection with pin loading
- Complex triggers
- Asynchronous data capture

Altera provides procedures to connect external verification devices such as oscilloscopes, logic analyzers, and protocol analyzers to your FPGA.

SignalProbe

The SignalProbe incremental routing feature allows you to route signals to output pins of the FPGA without affecting the existing fit of a design to a significant degree. You can use SignalProbe to investigate internal device signals without rewriting your HDL code to pass them up through multiple layers of the design hierarchy to a pin. Creating such revisions manually is time-consuming and error-prone.

Altera recommends SignalProbe when there are enough pins to route internal signals out of the FPGA for verification. If FPGA pins are not available, you have the following three alternatives:

- Reduce the number of pins used by the design to make more pins available to SignalProbe
- Use the SignalTap II embedded logic analyzer
- Use the Logic Analyzer Interface

Revising your design to increase the number of pins available for verification purposes requires design changes and can impact the design schedule. Using the SignalTap II embedded logic analyzer is a viable solution if you do not require continuous sampling at a high rate. The SignalTap II embedded logic analyzer does not require any additional pins to be routed; however, you must have enough unallocated logic and memory resources in your design to incorporate it. If neither of these approaches is viable, you can use the logic analyzer interface.

To learn more about SignalProbe, refer to the Quick Design Debugging Using SignalProbe chapter in the *Quartus Prime Handbook Volume 3: Verification*.

Related Information

Quick Design Debugging Using SignalProbe

Logic Analyzer Interface

The Quartus Prime Logic Analyzer Interface is a JTAG programmable method of driving multiple time-domain multiplexed signals to pins for external verification. Because the Logical Analyzer Interface multiplexes pins, it minimizes the pincount requirement. Groups of signals are assigned to a bank. Using JTAG as a communication channel, you can switch between banks.

You should use this approach when SignalTap II embedded logic analyzer is insufficient for your verification needs. Some external logic analyzer manufacturers support the Logic Analyzer Interface. These logic analyzers have various amounts of support. The most important feature is the ability to let the measurement tools cycle through the signal banks automatically.

The ability to cycle through signal banks is not limited to logic analyzers. You can use it for any external measurement tool. Some developers use low speed indicators, for example LEDs, for verification. You can use the Logic Analyzer interface to map many banks of signals to a small number of verification LEDs. You may wish to leave this form of verification in your final design so that your product is capable of creating low-level error codes after deployment.

To learn more about the Quartus Prime Logic Analyzer Interface, refer to the In-System Debugging Using External Logic Analyzers chapter in the *Quartus Prime Handbook Volume 3: Verification*.

Related Information

In-System Debugging Using External Logic Analyzers

Stimuli Generation

To effectively test your system you must maximize your test coverage with as few stimuli as possible. To maximize your test coverage you should use a combination of static and randomly generated data. The static data contains a fixed set of inputs that you can use to test the standard functionality and corner cases of your system.

Random tests are generated at run time, but must be accessible when failures occur so that you can analyze the failure case. Random test generation is particularly effective after static testing has identified the majority of issues with the basic functionality of your design. The test cases created may uncover unanticipated issues. Whenever randomly generated test inputs uncover issues with your system, you should add those cases to your static test data set for future testing.

Creating random data for use as inputs to your system can be challenging because pseudo random number generators (PRNG) tends to repeat patterns. Choose a different seed each time you initialize the PRNG for your random test generator. The random number generator creates the same data sequence if it is seeded with the same value.

Seed generation is an advanced topic and is not covered in detail in this section. The following recommendations on creating effective seed values should help you avoid repeating data values:

- Use a random noise measurement. One way to do this is by reading the analog output value of an A/D converter.
- Use multiple asynchronous counters in combination to create seed values.
- Use a timer value as the seed (that is, the number of seconds from a fixed point in time).

Using a combination of seed generation techniques can lead to more random behavior. When generating random sequences, it is important to understand the distribution of the random data generated. Some generators create linear sequences in which the distribution is evenly spread across the random number domain. Others create non-linear sequences that may not provide the test coverage you require. Before you begin using a random number generator to verify your system, examine the data created for a few sequences. Doing so helps you understand the patterns created and avoid using an inappropriate set of inputs.

Board Bring-up

You can minimize board bring-up time by adopting a systematic strategy. First, break the task down into manageable pieces. Verify the design in segments, not as a whole, beginning with peripheral testing.

Peripheral Testing

The first step in the board bring-up process is peripheral testing. Add one interface at a time to your design. After a peripheral passes the tests you have created for it, you should remove it from the test design. Designers typically leave the peripherals that pass testing in their design as they move on to test other peripherals. Sometimes this is necessary; however, it should be avoided when possible because multiple peripherals can create instability due to noise or crosstalk. By testing peripherals in a system individually, you can isolate the issues in your design to a particular interface.

A common failure in any system is involves memory. The most problematic memory devices operate at high speeds, which can result in timing failures. High performance memory also requires many board traces to transfer data, address, and control signals, which cause failures if not routed properly. You can use the Nios II processor to verify your memory devices using verification software. The Nios II processor is not capable of stress testing your memory but it can be used to detect memory address and data line issues.

For more information on debugging refer to the Debugging Nios II Designs chapter in this handbook.

Related Information

Debugging Nios II Designs on page 6-3

Data Trace Failure

If your board fabrication facility does not perform bare board testing, you must perform these tests. To detect data trace failures on your memory interface you should use a pattern typically referred to as "walking ones." The walking ones pattern shifts a logical 1 through all of the data traces between the FPGA and the memory device. The pattern can be increasing or decreasing; the important factor is that only one data signal is 1 at any given time. The increasing version of this pattern is as follows: 1, 2, 4, 8, 16, and so on.

Using this pattern you can detect a few issues with the data traces such as short or open circuit signals. A signal is short circuited when it is accidentally connected to another signal. A signal is open circuited when it is accidentally left unconnected. Open circuits can have a random signal behavior unless a pull-up or pull-down resistor is connected to the trace. If a pull-up or pull-down resistor is used, the signal drives a 0 or 1; however, the resistor is weak relative to a signal being driven by the test, so that test value overrides the pull-up or pull-down resistor.

To avoid mixing potential address and data trace issues in the same test, test only one address location at a time. To perform the test, write the test value out to memory, and then read it back. After verifying that the two values are equal, proceed to testing the next value in the pattern. If the verification stage detects a variation between the written and read values, a bit failure has occurred. The table below provides an example of the process used to find a data trace failure. It makes the simplifying assumption that sequential data bits are routed consecutively on the PCB.

Table 6-2: Walking Ones Example

Written Value	Read Value	Failure Detected
00000001	00000001	No failure detected
00000010	00000000	Error, most likely the second data bit, D[1] stuck low or shorted to ground
00000100	00000100	No failure detected, confirmed D[1] is stuck low or shorted to another trace that is not listed in this table.
00001000	00001000	No failure detected
00010000	00010000	No failure detected
00100000	01100000	Error, most likely D[6] and D[5] short circuited
01000000	01100000	Error, confirmed that D[6] and D[5] are short circuited
10000000	10000000	No failure detected

Address Trace Failure

The address trace test is similar to the walking ones test used for data with one exception. For this test you must write to all the test locations before reading back the data. Using address locations that are powers of two, you can quickly verify all the address traces of your circuit board.

The address trace test detects the aliasing effects that short or open circuits can have on your memory interface. For this reason it is important to write to each location with a different data value so that you can detect the address aliasing. You can use increasing numbers such as 1, 2, 3, 4, and so on while you verify the address traces in your system. The table below shows how to use powers of two in the process of finding an address trace failure:

Table 6-3: Powers of Two Example

Address	Written Value	Read Value	Failure Detected
00000000	1	1	No failure detected
00000001	2	2	No failure detected
00000010	3	3	Error, the second address bit, A[1], is stuck low
00000100	4	4	No failure detected

Address	Written Value	Read Value	Failure Detected
00001000	5	5	No failure detected
00010000	6	6	No failure detected
00100000	7	7	Error, A[5] and A[4] are short circuited
01000000	8	8	No failure detected
10000000	9	9	No failure detected

Device Isolation

Using device isolation techniques, you can disable features of devices on your PCB that cause your design to fail. Typically designers use device isolation for early revisions of the PCB, and then remove these capabilities before shipping the product.

Most designs use crystal oscillators or other discrete components to create clock signals for the digital logic. If the clock signal is distorted by noise or jitter, failures may occur. To guard against distorted clocks, you can route alternative clock pins to your FPGA. If you include SMA connectors on your board, you can use an external clock generator to create a clean clock signal. Having an alternative clock source is very useful when debugging clock-related issues.

Sometimes the noise generated by a particular device on your board can cause problems with other devices or interfaces. Having the ability to reduce the noise levels of selected components can help you determine the device that is causing issues in your design. The simplest way to isolate a noisy component is to remove the power source for the device in question. For devices that have a limited number of power pins, if you include 0 ohm resistors in the path between the power source and the pin. You can cut off power to the device by removing the resistor. This strategy is typically not possible with larger devices that contain multiple power source pins connecting directly to a board power plane.

Instead of removing the power source from a noisy device, you can often put the device into a reset state by driving the reset pin to an active state. Another option is to simply not exercise the device so that it remains idle.

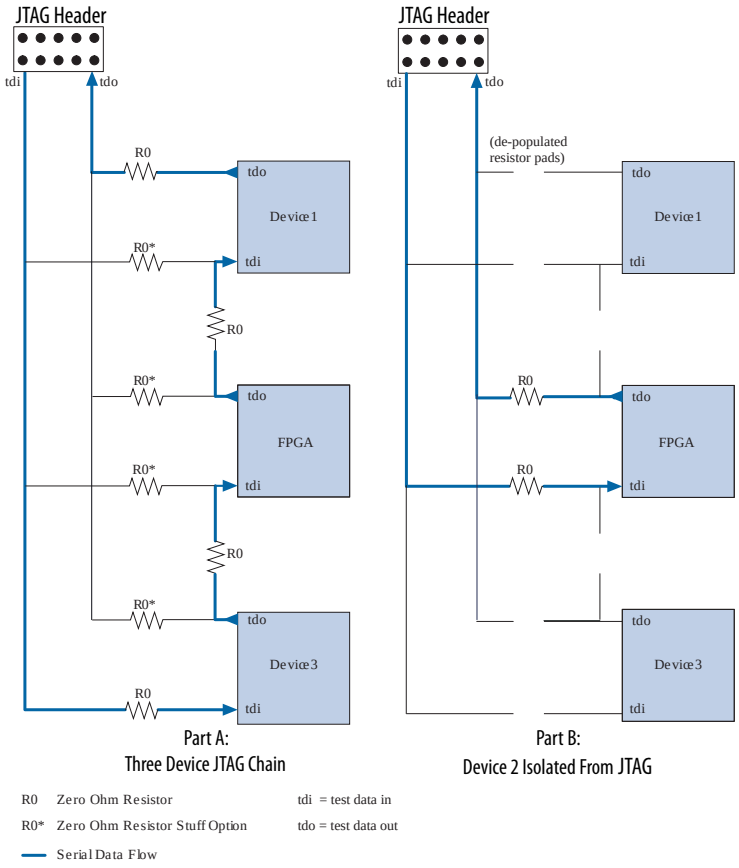
A noisy power supply or ground plane can create signal integrity issues. With the typical voltage swing of digital devices frequently below a single volt, the power supply noise margin of devices on the PCB can be as little as 0.2 volts. Power supply noise can cause digital logic to fail. For this reason it is important to be able to isolate the power supplies on your board. You can isolate your power supply by using fuses that are removed so that a stable external power supply can be substituted temporarily in your design.

JTAG

FPGAs use the JTAG interface for programming, communication, and verification. Designers frequently connect several components, including FPGAs, discrete processors, and memory devices, communicating with them through a single JTAG chain. Sometimes the JTAG signal is distorted by electrical noise, causing a communication failure for the entire group of devices. To guarantee a stable connection, you must isolate the FPGA under test from the other devices in the same JTAG chain.

The figure below **Part A** illustrates a JTAG chain with three devices. The tdi and tdo signals include 0 ohm resistors between each device. By removing the appropriate resistors, it is possible to isolate a single device in the chain as the figure below **Part B** illustrates. This technique allows you to isolate one device while using a single JTAG chain.

Figure 6-9: JTAG Isolation



To learn more about JTAG refer to *AN39: IEEE 1149.1(JTAG) Boundary-Scan Testing in Altera Devices*.

Related Information

IEEE 1149.1 JTAG Boundary-Scan Testing in Altera Devices

Board Testing

You should convert the simulations you run to verify your intellectual property (IP) before fabrication to test vectors that you can then run on the hardware to verify that the simulation and hardware versions exhibit the same behavior. Manufacturing can also use these tests as part of a regularly scheduled quality assurance test. Because the tests are run by engineers in other organizations they must be documented and easy to run.

Minimal Test System

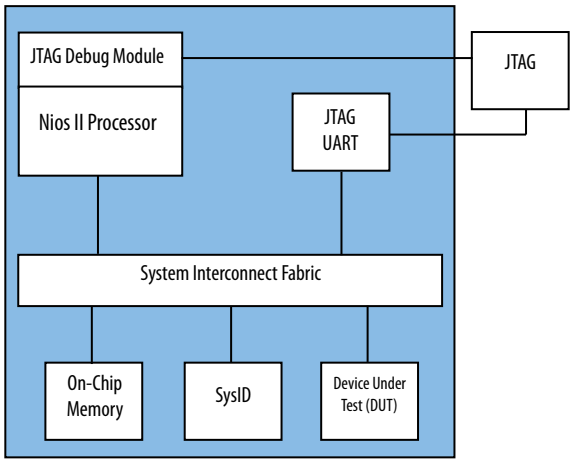
Whether you are creating your first embedded system in a FPGA, or are debugging a complex issue, you should always begin with a minimal system. To minimize the probability of signal integrity issues, reduce the pincount of your system to the absolute minimal number of required pins. In an embedded design that

includes the Nios II processor, the minimal pincount might be clock and reset signals. Such a system might include the following the following components:

- Nios II processor (with a level 1 debug core)
- On-chip memory
- JTAG UART
- System ID core

Using these four components you can create a functioning embedded system including debug and terminal access. To simplify your debug process, you should use a Nios II processor that does not contain a data cache. The Nios II/e core does not include data caches. The Nios II/f core can also be configured without a data cache. The figure below illustrates a minimal system. In this system, you have to route only the clock pin and reset pins, because the JTAG signals are automatically connected by the Quartus Prime software.

Figure 6-10: Simple Test System



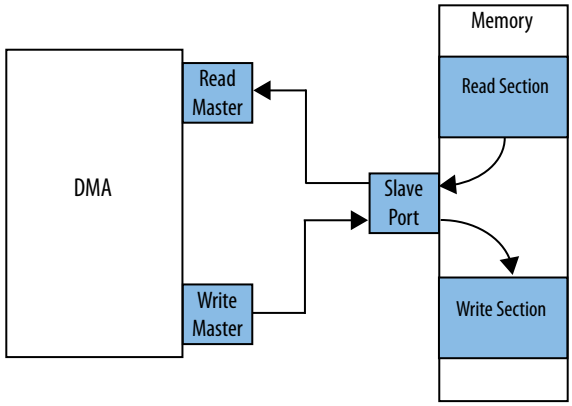
You can use the Nios II JTAG debug module to download software to the processor. Before testing any additional interfaces you should execute a small program that prints a message to the terminal to verify that your minimal system is functioning properly.

After you verify that the simple test system functions properly, archive the design. This design provides a stable starting point to which to add additional components as verification proceeds. In this system, you can use any of the following for testing:

- A Nios II processor
- The SignalTap II embedded logic analyzer
- An external logic interface
- SignalProbe
- A direct memory access (DMA) engine
- In-system updating of memory and constants

The Nios II processor is not capable of stress testing high speed memory devices. Altera recommends that you use a DMA engine to stress test memories. A stress test should access memory as frequently as possible, performing continuous reads or writes. Typically, the most problematic access sequence for high-speed memory involves the bus turnaround between read and write accesses. You can test these cases by connecting the DMA read and write masters to the same memory and transferring the contents from one location to another, as shown below.

Figure 6-11: Using a DMA to Stress Test Memory Devices



By modifying the arbitration share values for each master to memory connection, you can control the sequence. To alternate reads and writes, you can use an arbitration share of one for each DMA master port. To perform two reads followed by two writes, use an arbitration value of two for each DMA master port. To create more complicated access sequences you can create a custom master.

Related Information

- **Nios II Hardware Development Tutorial**
- **In-System Modification of Memory and Constants**
- **Quartus Prime Software: Verification**
- **DMA Controller**

System Verification

System verification is the final step of system design. This section focuses on common mistakes designers make during system verification and methods for correcting and avoiding them. It includes the following topics:

- Designing with Verification in Mind
- Accelerating Verification
- Using Software to Verify Hardware
- Environmental Testing

Designing with Verification in Mind

As you design, you should focus on both the development tasks and the verification strategy. Doing so results in a design that is easier to verify. If you create large, complicated blocks of logic and wait until the HDL code is complete before testing, you spend more time verifying your design than if you verify it one section at a time.

Consider leaving in verification code after the individual sections of your design are working. If you remove too much verification logic it becomes very difficult to reintroduce it at a later time if needed. If you discover an issue during system integration, you may need to revisit some of the smaller block designs. If you modify one of the smaller blocks, you must re-test it to verify that you have not created additional issues.

Designing with verification in mind is not limited to leaving verification hooks in your design. Reserving enough hardware resources to perform proper verification is also important. The following recommendations can help you avoid running out of hardware resources:

- Design and verify using a larger pin-compatible FPGA.
- Reserve hardware resources for verification in the design plan.
- Design the logic so that optional features can be removed to free up verification resources.

Finally, schedule a nightly regression test of your design to increase your test coverage between hardware or software compilations.

Accelerating Verification

Altera recommends the verification flow illustrated in the figure below. Verify each component as it is developed. By minimizing the amount of logic being verified, you can reduce the time it takes to compile and simulate your design. Consequently, you minimize the iteration time to correct design issues.

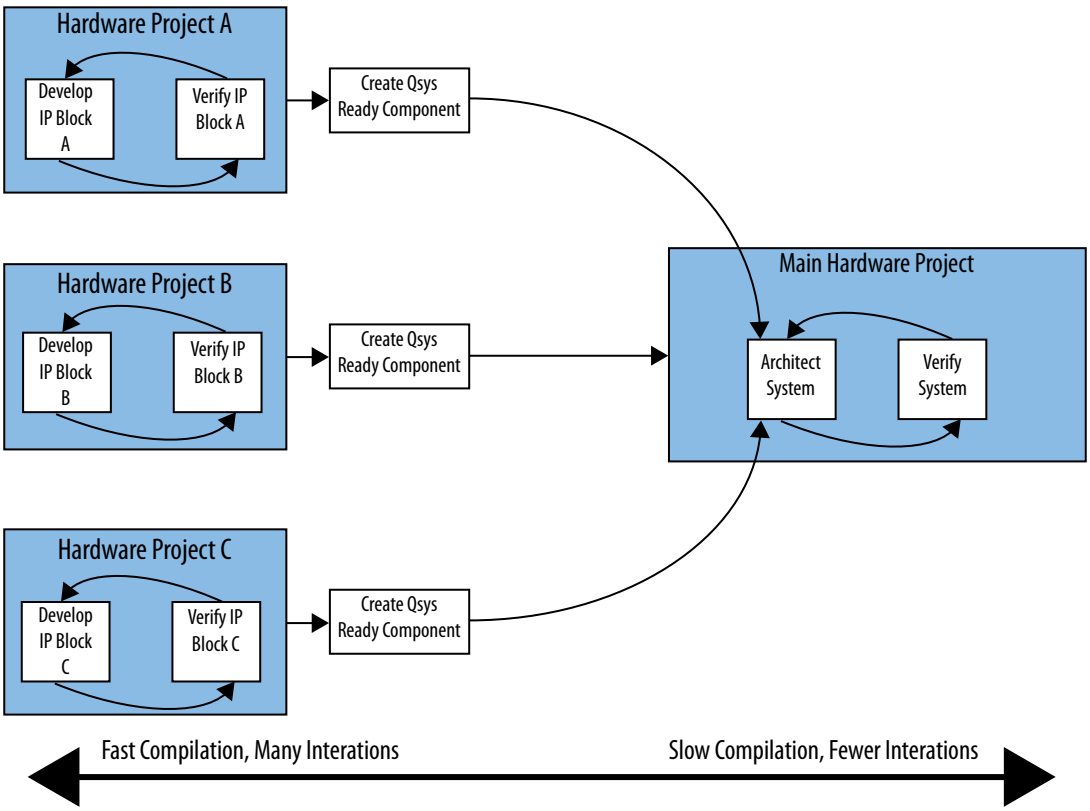
After the individual components are verified, you can integrate them in a Qsys system. The integrated system must include an Avalon-MM or Avalon Streaming (Avalon-ST) port. Using the component editor available from Qsys, you add an Avalon-MM interface to your existing component and integrate it in your system.

After your system is created in Qsys, you can continue the verification process of the system as a whole. Typically, the verification process has the following two steps:

1. Generate then simulate
2. Generate, compile, and then verify in hardware

The first step provides easier access to the signals in your system. When the simulation is functioning properly, you can move the verification to hardware. Because the hardware is orders of magnitude faster than the simulation, running test vectors on the actual hardware saves time.

Figure 6-12: IP Verification and Integration Flow



To learn more about component editor and system integration, refer to the following documentation:

Related Information

- **Creating Qsys Components**
- **Avalon Interface Specifications**

Using Software to Verify Hardware

Many hardware developers use test benches and test harnesses to verify their logic in simulations. These strategies can be very time consuming. Instead of relying on simulations for all your verification tasks, you can test your logic using software or scripts, as the figure below illustrates.

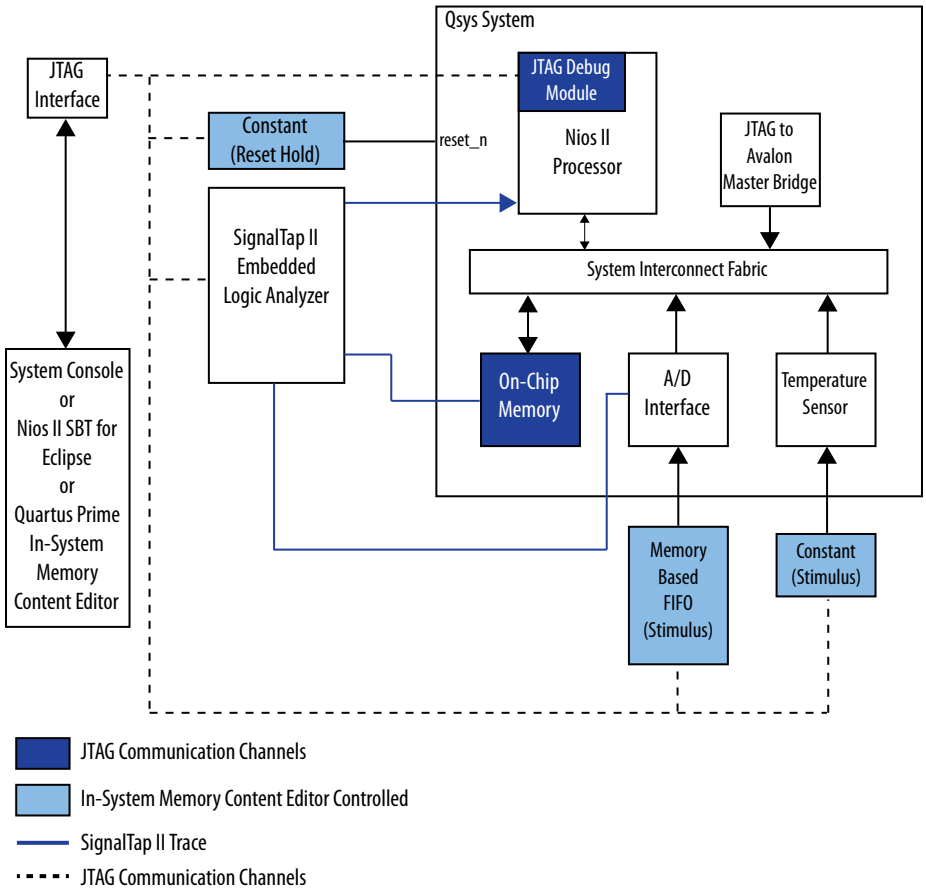
This system uses the JTAG interface to access components connected to the system interconnect fabric and to create stimuli for the system. If you use the JTAG server provided by the Quartus Prime programmer, this system can also be located on a network and accessed remotely. You can download software to the Nios II processor using the Nios II SBT. You can also use the Nios II JTAG debug core to transmit files to and from your embedded system using the host file system. Using the System Console you can access components in your system and also run scripts for automated testing purposes.

Using the Quartus Prime In-System Memory Content Editor, you can create stimuli for the two components that control external peripherals. You can also use the In-System Memory Content Editor to place the embedded system in reset while new stimulus values are sent to the system. The In-System Memory Editor supports Tcl scripting, which you can use to automate the verification process. All of the

verification techniques described in this section can be scripted, allowing many test cycles to be executed without user interaction.

To learn more about using the host file system refer to the Host File System software example design available with the Nios II EDS. The Developing Nios II Software chapter of the Embedded Design Handbook also includes a significant amount of information about the system file system.

Figure 6-13: Script Controlled Verification



To learn more about the verification and scripting abilities outlined in the example above, refer to the following documentation:

Related Information

Tcl Scripting

Environmental Testing

The last stage of verification is end-user environment testing. Most verification is performed under ideal conditions. The following conditions in the end user's environment can cause the system to fail:

- Voltage variation
- Vibration
- Temperature variation
- Electrical noise

Because it is difficult to predict all the applications for a particular product, you should create a list of operational specifications before designing the product. You should verify these specifications before shipping or selling the product. The key issue with environmental testing is the difficulty associated with obtaining measurements while the test is underway. For example, it can be difficult to measure signals with an external logic analyzer while your product is undergoing vibration testing.

While choosing methods to test your hardware design during the early verification stages, you should also consider how to adapt them for environmental testing. If you believe your product is susceptible to vibration problems, you should choose sturdy instrumentation methods when testing memory interfaces. Alternatively, if you believe your product may be susceptible to electrical noise, then you should choose a highly reliable interface for debug purposes.

While performing early verification of your design, you can also begin end-environment testing. Doing so helps you detect potential flaws in early in the design process. For example, if you wish to test temperature variations, you can use a heat gun on the product while you are testing. If you wish to perform voltage variation testing, isolate the power supply in your system and vary the voltage using an external power supply. Using these verification techniques, you can avoid late design changes due to failures during environmental testing.

Additional Embedded Design Considerations

Consider the following topics as you design your system:

- JTAG signal integrity
- Extra memory space for prototyping
- System verification

JTAG Signal Integrity

The JTAG signal integrity on your system is very important. You must debug your hardware and software, and program your FPGA, through the JTAG interface. Poor signal integrity on the JTAG interface can prevent you from debugging over the JTAG connection, or cause inconsistent debugger behavior.

You can use the System Console to verify the JTAG chain.

JTAG signal integrity problems are extremely difficult to diagnose. To increase the probability of avoiding these problems, and to help you diagnose them should they arise, Altera recommends that you follow the guidelines outlined in AN428: MAX II CPLD Design Guidelines and in the Verification and Board Bring-Up chapter of this handbook when designing your board.

Note: For more information about the System Console, refer to the Analyzing and Debugging Designs with the System Console chapter in volume 3 of the Quartus Prime Handbook.

Related Information

- **AN428: MAX II CPLD Design Guidelines**
- **Analyzing and Debugging Designs with System Console**

Memory Space For System Prototyping

Even if your final product includes no off-chip memory, Altera recommends that your prototype board include a connection to some region of off-chip memory. This component in your system provides additional memory capacity that enables you to focus on refining code functionality without worrying about code size. Later in the design process, you can substitute a smaller memory device to store your software.

Document Revision History

Table 6-4: Nios II Debug, Verification, and Simulation Chapter Revision History

Date	Version	Changes
December 2016	2016.12.19	Initial release.

Optimizing Nios II Based Systems and Software

7

2016.12.19

ED_HANDBOOK Subscribe Send Feedback

Optimizing techniques enable better performance and resource utilization in your design. This chapter provides various optimizing techniques from the hardware and software perspective.

Hardware Acceleration and Coprocessing

This section discusses how you can use hardware accelerators and coprocessing to create more efficient, higher throughput designs in Qsys. This section discusses the following topics:

- Accelerating Cyclic Redundancy Checking (CRC)
- Creating Nios II Custom Instructions
- Creating Multicore Designs
- Pre- and Post-Processing
- Replacing State Machines

Hardware Acceleration

Hardware accelerators implemented in FPGAs offer a scalable solution for performance-limited systems. Other alternatives for increasing system performance include choosing higher performance components or increasing the system clock frequency. Although these other solutions are effective, in many cases they lead to additional cost, power, or design time.

Customizing and Accelerating FPGA Designs

FPGA-based designs provide you with the flexibility to modify your design easily, and to experiment to determine the best balance between hardware and software implementation of your design. In a discrete microcontroller-based design process, you must determine the processor resources—cache size and built-in peripherals, for example—before you reach the final design stages. You may be forced to make these resource decisions before you know your final processor requirements. If you implement some or all of your system's critical design components in an FPGA, you can easily redesign your system as your final product needs become clear. If you use the Nios II processor, you can experiment with the correct balance of processor resources to optimize your system for your needs. Qsys facilitates this flexibility, by allowing you to add and modify system components and regenerate your project easily.

To experiment with performance and resource utilization tradeoffs, the following hardware optimization techniques are available:

- **Processor Performance**—You can increase the performance of the Nios II processor in the following ways:
  - **Computational Efficiency**—Selecting the most computationally efficient Nios II processor core is the quickest way to improve overall application performance. The following Nios II processor cores are available, in decreasing order of performance:(6)
    - Nios II/f—optimized for speed
    - Nios II/e—conserves on-chip resources at the expense of speed
  - **Memory Bandwidth**—Using low-latency, high speed memory decreases the amount of time required by the processor to fetch instructions and move data. Additionally, increasing the processor's arbitration share of the memory increases the processor's performance by allowing the Nios II processor to perform more transactions to the memory before another Avalon master port can assume control of the memory.
  - **Instruction and Data Caches**—Adding an instruction and data cache is an effective way to decrease the amount of time the Nios II processor spends performing operations, especially in systems that have slow memories, such as SDRAM or double data rate (DDR) SDRAM. In general, the larger the cache size selected for the Nios II processor, the greater the performance improvement.
  - **Tightly-coupled Memories**—Tightly-coupled memory is fast on-chip memory that bypasses the cache and has guaranteed low latency. Tightly-coupled memory gives the best memory access performance. You assign code and data to tightly-coupled memory partitions in the same way as other memory sections.
  - **Hardware Multipliers**—The Nios II processor provides the following hardware multiplier options:
    - DSP Block—Includes DSP block multipliers available on the target device. This option is available only on Altera FPGAs that have DSP Blocks.
    - Embedded Multipliers—Includes dedicated embedded multipliers available on the target device. This option is available only on Altera FPGAs that have embedded multipliers.
    - Logic Elements—Includes hardware multipliers built from logic element (LE) resources.
    - None—Does not include multiply hardware. In this case, multiply operations are emulated in software
  - **Optional Branch Prediction**—The Nios II processor performs dynamic and static branch prediction to minimize the cycle penalty associated with taken branches.
- **Clock Frequency**—Increasing the speed of the processor's clock results in more instructions being executed per unit of time. To gain the best performance possible, ensure that the processor's execution memory is in the same clock domain as the processor, to avoid the use of clock-crossing FIFO buffers.

  One of the easiest ways to increase the operational clock frequency of the processor and memory peripherals is to use a FIFO bridge IP core to isolate the slower peripherals of the system. With a bridge peripheral, for example, you can connect the processor, memory, and an Ethernet device on one side of the bridge, and connect all of the peripherals that are not performance dependent on the other side of the bridge.

Similarly, if you implement your system in an FPGA, you can experiment with the best balance of hardware and software resource usage. If you find you have a software bottleneck in some part of your application, you can consider accelerating the relevant algorithm by implementing it in hardware instead of software. Qsys facilitates experimenting with the balance of software and hardware implementation. You can even design custom hardware accelerators for specific system tasks.

(6) The Nios II/s core is only available with Nios II Classic.

To help you solve system performance issues, the following acceleration methodologies are available:

- Custom peripherals
- Custom instructions

The method of acceleration you choose depends on the operation you wish to accelerate. To accelerate streaming operations on large amounts of data, a custom peripheral may be a good solution. Hardware interfaces (such as implementations of the Ethernet or serial peripheral interface (SPI) protocol) may also be implemented efficiently as custom peripherals. The current floating-point custom instruction is a good example of the type of operations that are typically best accelerated using custom instructions.

For information about hardware acceleration, refer to the "Hardware Acceleration and Coprocessing" chapter of the *Embedded Design Handbook*.

For information about custom instructions, refer to the *Nios II Custom Instruction User Guide*.

For information about creating custom peripherals, refer to the "Creating Qsys Components" chapter in the *Quartus Prime Handbook Volume 1: Design and Synthesis*.

Related Information

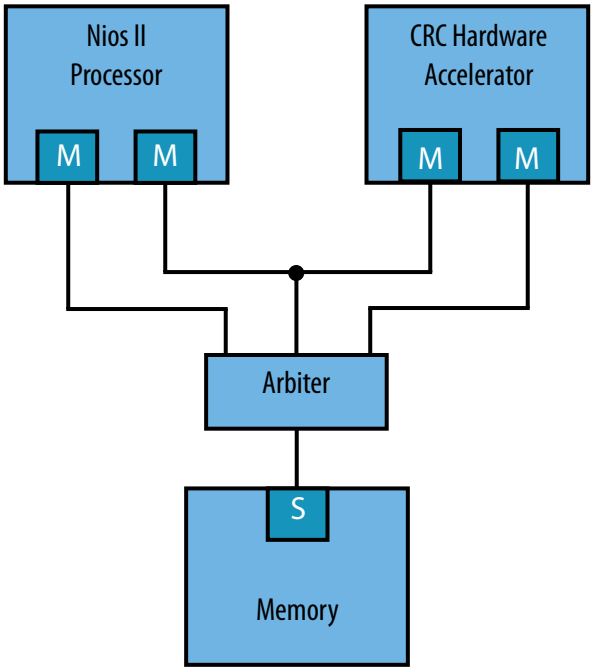
- **Creating Qsys Components**
- **Nios II Custom Instruction User Guide**
- **Hardware Acceleration and Coprocessing** on page 7-1

Accelerating Cyclic Redundancy Checking (CRC)

CRC is significantly more efficient in hardware than software; consequently, you can improve the throughput of your system by implementing a hardware accelerator for CRC. In addition, by eliminating CRC from the tasks that the processor must run, the processor has more bandwidth to accomplish other tasks.

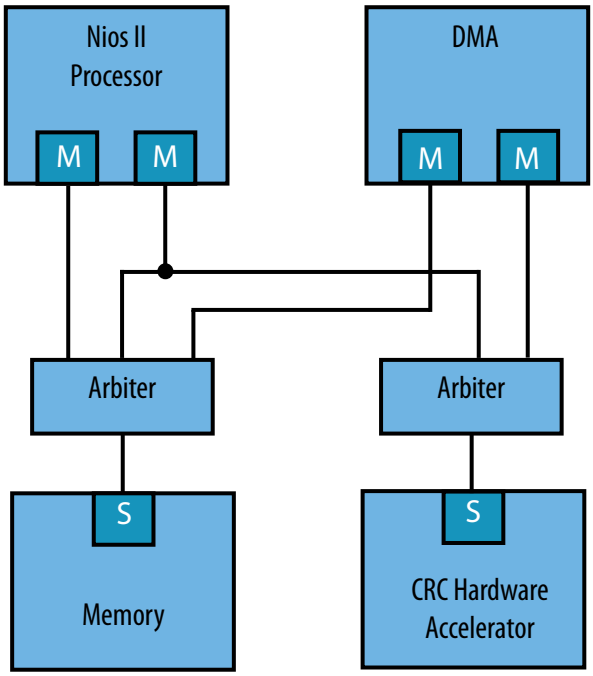
The figure below illustrates a system in which a Nios II processor offloads CRC processing to a hardware accelerator. In this system, the Nios II processor reads and writes registers to control the CRC using its Avalon Memory-Mapped (Avalon-MM) slave port. The CRC component's Avalon-MM master port reads data for the CRC check from memory.

Figure 7-1: A Hardware Accelerator for CRC



An alternative approach includes a dedicated DMA engine in addition to the Nios II processor. The figure below illustrates this design. In this system, the Nios II processor programs the DMA engine, which transfers data from memory to the CRC.

Figure 7-2: DMA and Hardware Accelerator for CRC



Although the figure above shows the DMA and CRC as separate blocks, you can combine them as a custom component which includes both an Avalon-MM master and slave port. You can import this component into your Qsys system using the component editor.

To learn more about using component editor, refer to the "Component Editor" section in the Creating Qsys Components chapter of the *Quartus Prime Handbook Volume 1:Design and Synthesis*. You can find additional examples of hardware acceleration on the Altera IP:Reference Designs web page.

Related Information

- **Altera IP: Reference Designs**
- **Creating Qsys Components**

Matching I/O Bandwidths

I/O bandwidth can have a large impact on overall performance. Low I/O bandwidth can cause a high-performance hardware accelerator to perform poorly when the dedicated hardware requires higher throughput than the I/O can support. You can increase the overall system performance by matching the I/O bandwidth to the computational needs of your system.

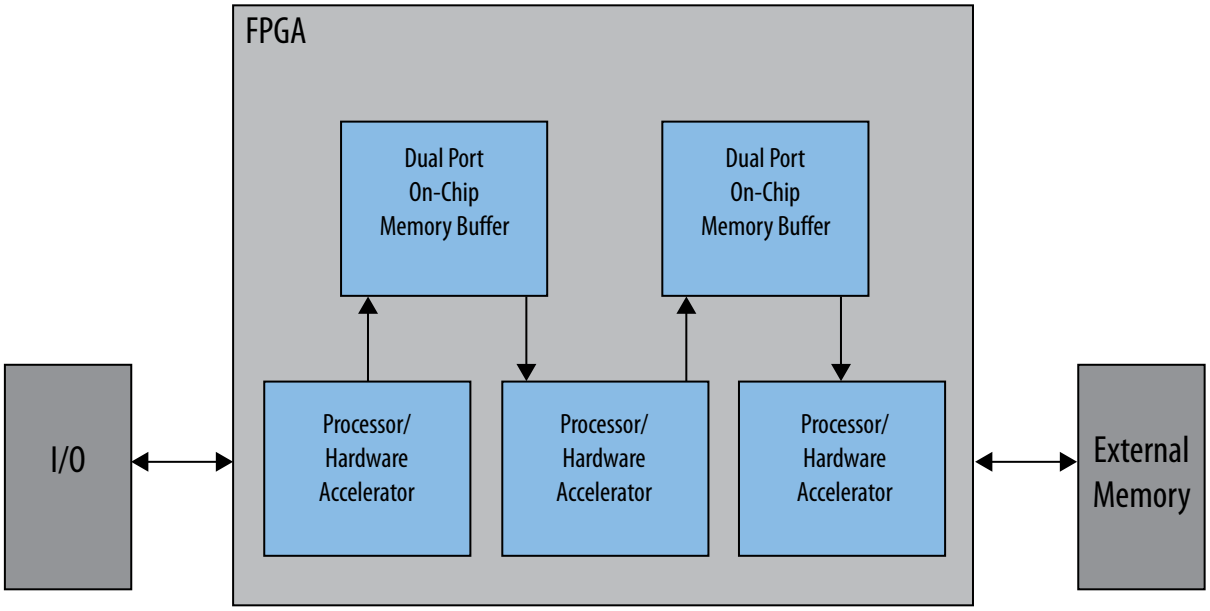
Typically, memory interfaces cause the most problems in systems that contain multiple processors and hardware accelerators. The following recommendations for interface design can maximize the throughput of your hardware accelerator:

- Match high performance memory and interfaces to the highest priority tasks your system must perform.
- Give high priority tasks a greater share of the I/O bandwidth if any memory or interface is shared.
- If you have multiple processors in your system, but only one of the processors provides real-time functionality, assign it a higher arbitration share.

Pipelining Algorithms

A common problem in systems with multiple Avalon-MM master ports is competition for shared resources. You can improve performance by pipelining the algorithm and buffering the intermediate results in separate on-chip memories. The figure below illustrates this approach. Two hardware accelerators write their intermediate results to on-chip memory. The third module writes the final result to an off-chip memory. Storing intermediate results in on-chip memories reduces the I/O throughput required of the off-chip memory. By using on-chip memories as temporary storage you also reduce read latency because on-chip memory has a fixed, low-latency access time.

Figure 7-3: Using On-Chip Memory to Achieve High Performance



To learn more about optimizing memory design refer to the Memory System Design chapter of this handbook.

Related Information

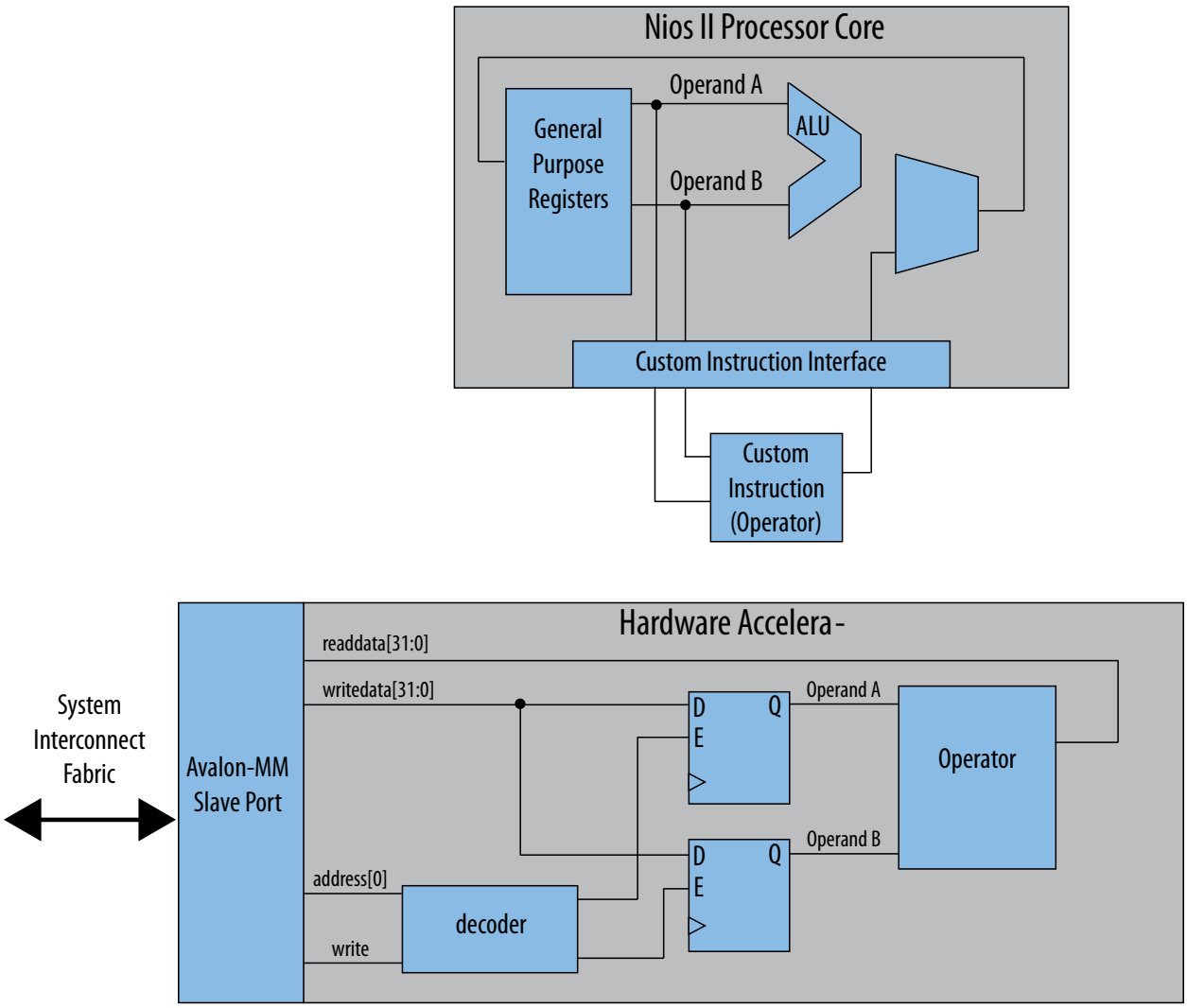
Memory System Design on page 3-35

Creating Nios II Custom Instructions

The Nios II processor employs a RISC architecture which can be expanded with custom instructions. The Nios II processor includes a standard interface that you can use to implement your own custom instruction hardware in parallel with the arithmetic logic unit (ALU).

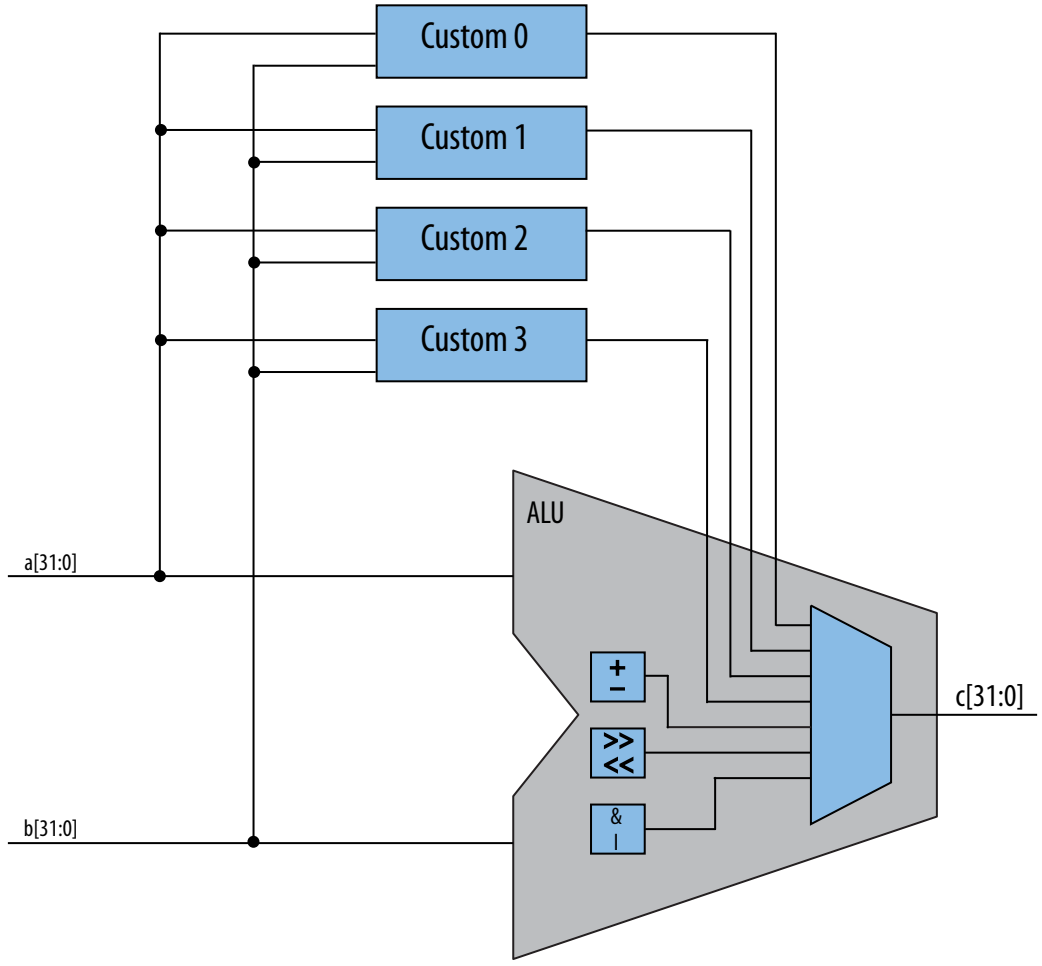
All custom instructions have a similar structure. They include up to two data inputs and one data output, and optional clock, reset, mode, address, and status signals for more complex multicycle operations. If you need to add hardware acceleration that requires many inputs and outputs, a custom hardware accelerator with an Avalon-MM slave port is a more appropriate solution. Custom instructions are blocking operations that prevent the processor from executing additional instructions until the custom instruction has completed. To avoid stalling the processor while your custom instruction is running, you can convert your custom instruction into a hardware accelerator with an Avalon-MM slave port. If you do so, the processor and custom peripheral can operate in parallel. The differences in implementation between a custom instruction and a hardware accelerator are illustrated below.

Figure 7-4: Implementation Differences between a Custom Instruction and Hardware Accelerator



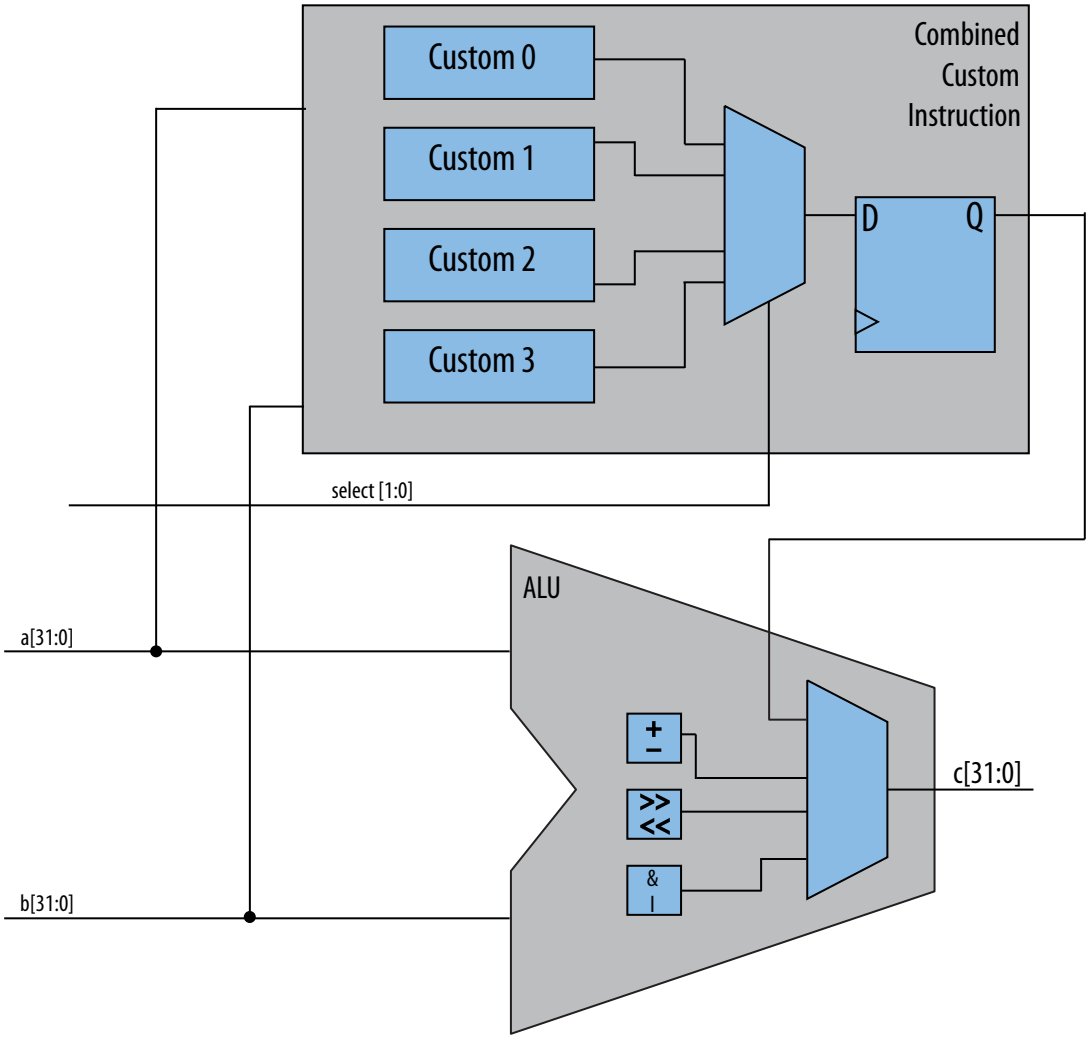
Because custom instructions extend the Nios II processor's ALU, the logic must meet timing or the fMAX of the processor will suffer. As you add custom instructions to the processor, the ALU multiplexer grows in width as the figure below illustrates. This multiplexer selects the output from the ALU hardware (c[31:0]). Although you can pipeline custom instructions, you have no control over the automatically inserted ALU multiplexer. As a result, you cannot pipeline the multiplexer for higher performance.

Figure 7-5: Individual Custom Instructions



Instead of adding several custom instructions, you can combine the functionality into a single logic block as shown in the "Combined Custom Instruction" figure below. When you combine custom instructions you use selector bits to select the required functionality. If you create a combined custom instruction, you must insert the multiplexer in your logic manually. This approach gives you full control over the multiplexer logic that generates the output. You can pipeline the multiplexer to prevent your combined custom instruction from becoming part of a critical timing path.

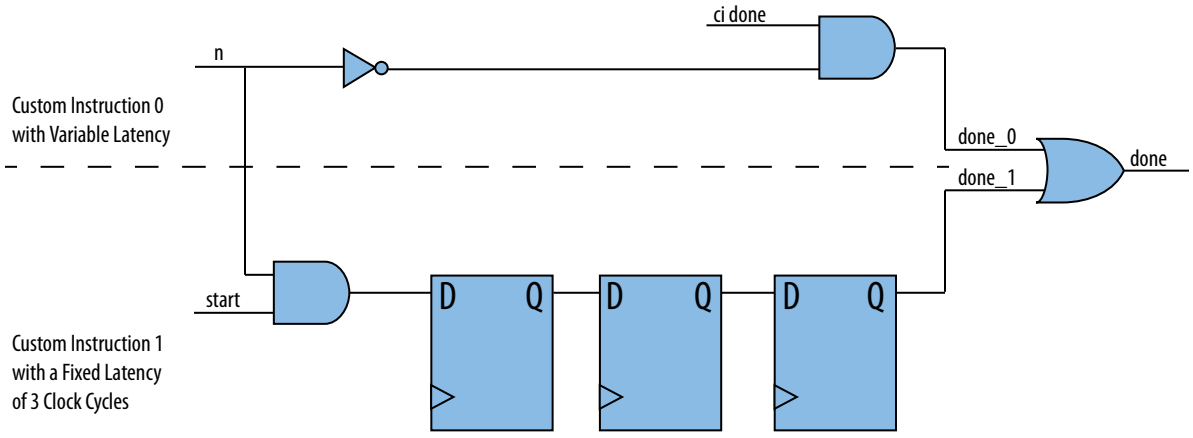
Figure 7-6: Combined Custom Instruction



With multiple custom instructions built into a logic block, you can pipeline the output if it fails timing. To combine custom instructions, each must have identical latency characteristics.

Custom instructions are either fixed latency or variable latency. You can convert fixed latency custom instructions to variable latency by adding timing logic. The figure below shows the simplest method to implement this conversion by shifting the start bit by <n> clock cycles and logically ORing all the done bits.

Figure 7-7: Sequencing Logic for Mixed Latency Combined Custom Instruction



Each custom instruction contains at least one custom instruction slave port, through which it connects to the ALU. A custom instruction slave is that slave port: the slave interface that receives the data input signals and returns the result. The custom instruction master is the master port of the processor that connects to the custom instruction.

For more information about creating and using custom instructions refer to the *Nios II Custom Instruction User Guide*.

Related Information

Nios II Custom Instruction User Guide

Coprocessing

Partitioning system functionality between a Nios II processor and hardware accelerators or between multiple Nios II processors in your FPGA can help you control costs. The following sections demonstrate how you can use coprocessing to create high performance systems.

Creating Multicore Designs

Multicore designs combine multiple processor cores in a single FPGA to create a higher performance computing system. Typically, the processors in a multicore design can communicate with each other. Designs including the Nios II processor can implement inter-processor communication, or the processors can operate autonomously.

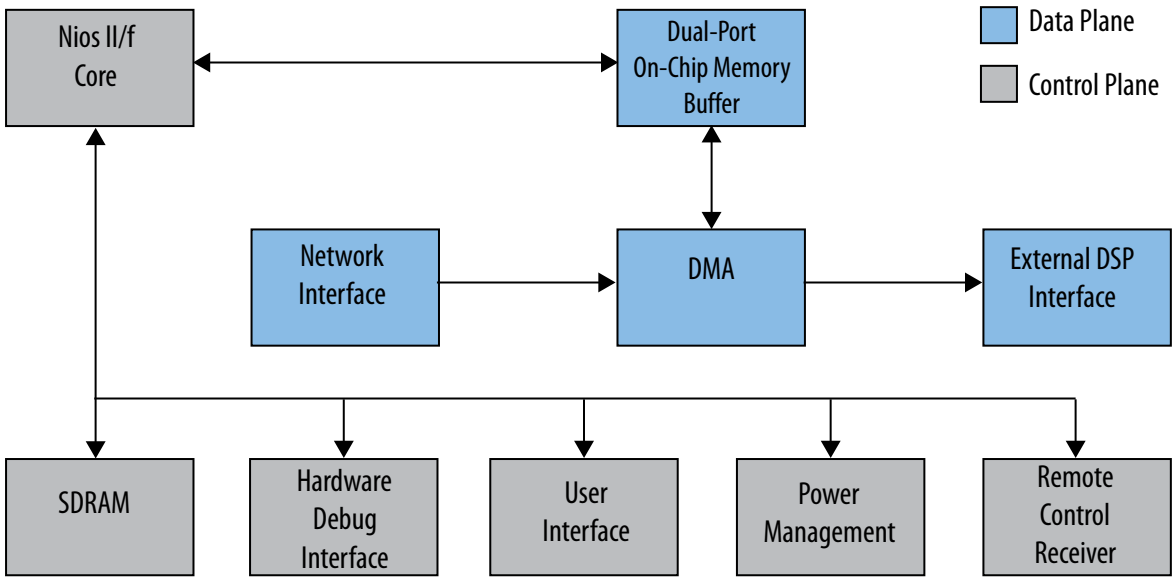
When a design includes more than one processor you must partition the algorithm carefully to make efficient use of all of the processors. The following example includes a Nios II-based system that performs video over IP, using a network interface to supply data to a discrete DSP processor. The original design overutilizes the Nios II processor. The system performs the following steps to transfer data from the network to the DSP processor:

1. The network interface signals when a full data packet has been received.
2. The Nios II processor uses a DMA engine to transfer the packet to a dual-port on-chip memory buffer.
3. The Nios II processor processes the packet in the on-chip memory buffer.
4. The Nios II processor uses the DMA engine to transfer the video data to the DSP processor.

In the original design, the Nios II processor is also responsible for communications with the following peripherals that include Avalon-MM slave ports:

- Hardware debug interface
- User interface
- Power management
- Remote control receiver

Figure 7-8: Over-Utilized Video System

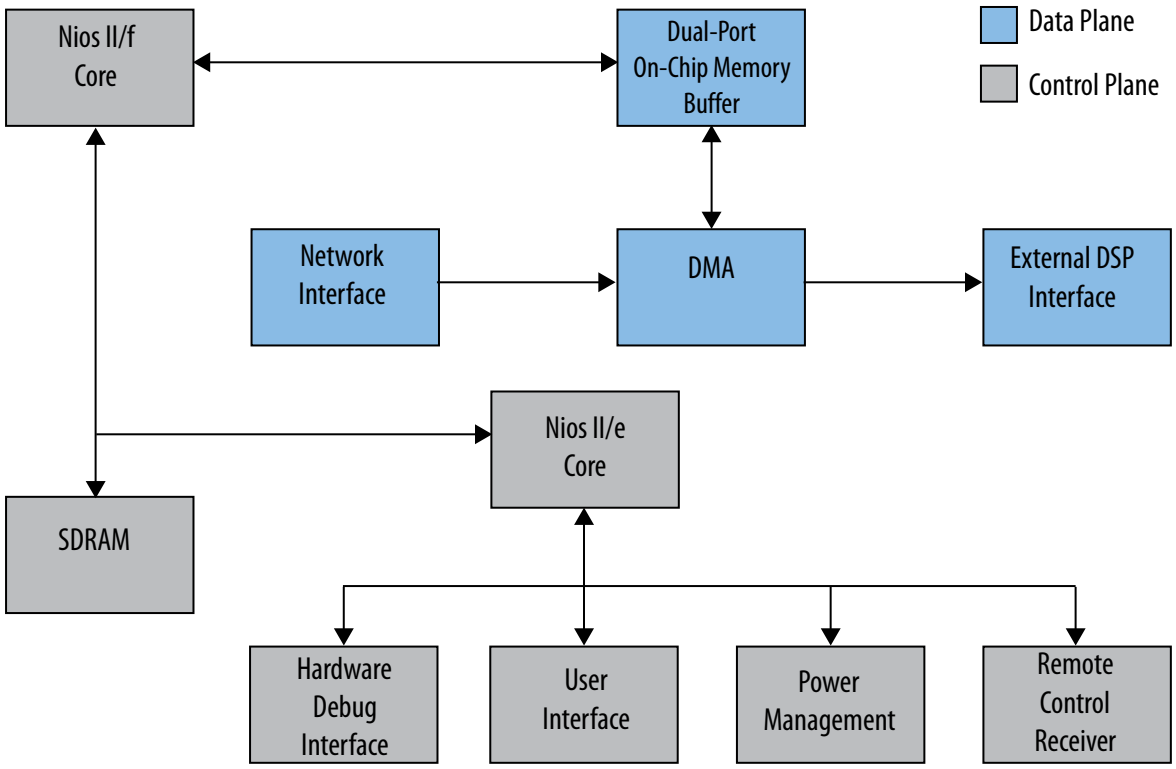


Adding a second Nios II processor to the system, allows the workload to be divided so that one processor handles the network functionality and the other the control interfaces.

Because the revised design has two processors, you must create two software projects; however, each of these software projects handles fewer tasks and is simpler to create and maintain. You must also create a mechanism for inter-processor communication. The inter-processor communication in this system is relatively simple and is justified by the system performance increase.

For more information about designing hardware and software for inter-processor communication, refer to the *Creating Multiprocessor Nios II Systems Tutorial* and the Peripherals section of the *Embedded Peripherals IP User Guide*. Refer to the *Nios II Gen2 Processor Reference Handbook* for complete information about this soft core processor. A Nios II Multiprocessor Design Example is available on the Altera website.

Figure 7-9: High Performance Video System



In the figure above, the second Nios II processor added to the system performs primarily low-level maintenance tasks; consequently, the Nios II/e core is used. The Nios II/e core implements only the most basic processor functionality in an effort to trade off performance for a small hardware footprint. This core is approximately one-third the size of the Nios II/f core.

To learn more about the three Nios II processor cores refer to the Nios II Core Implementation Details chapter in the *Nios II Gen2 Processor Reference Handbook*.

Related Information

- **Creating Multiprocessor Nios II Systems Tutorial**
- **Embedded Peripherals IP User Guide**
- **Nios II Core Implementation Details**
- **Nios II Multiprocessor Design Example**

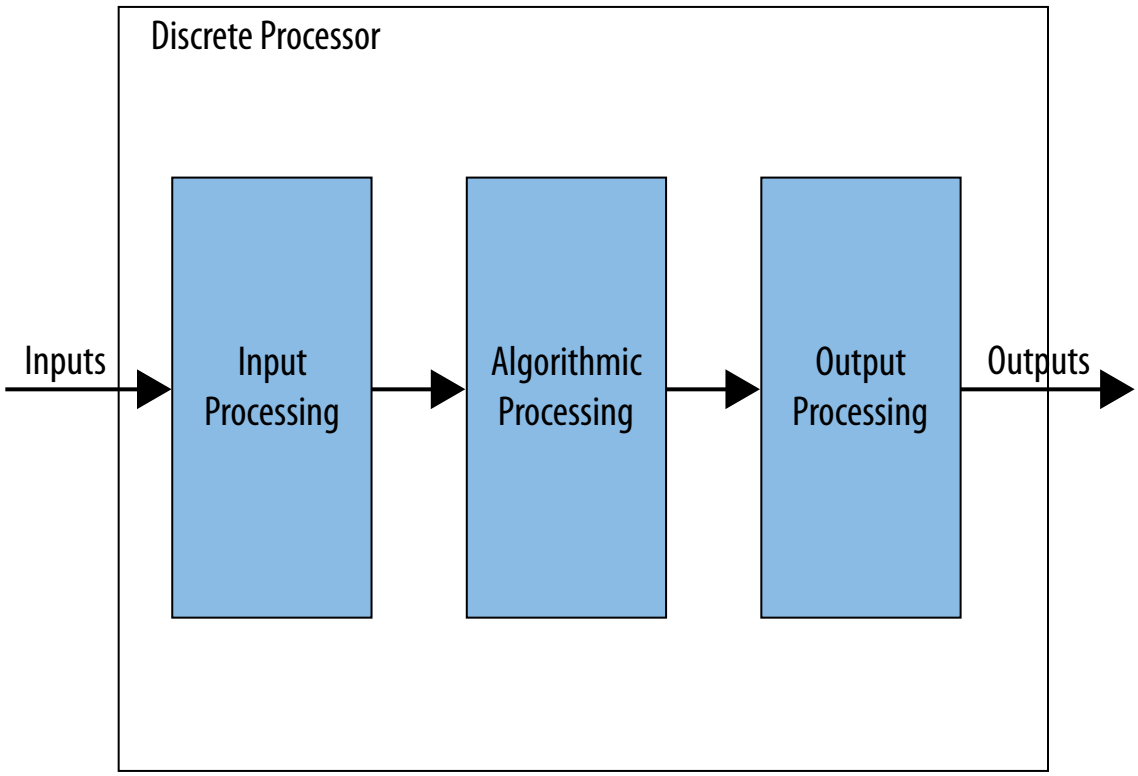
Pre- and Post-Processing

The high performance video system illustrated in **Figure 7-9** distributes the workload by separating the control and data planes in the hardware. **Figure 7-11** illustrates a different approach. All three stages of a DSP workload are implemented in software running on a discrete processor. This workload includes the following stages:

- Input processing—typically removing packet headers and error correction information
- Algorithmic processing and error correction—processing the data
- Output processing—typically adding error correction, converting data stream to packets, driving data to I/O devices

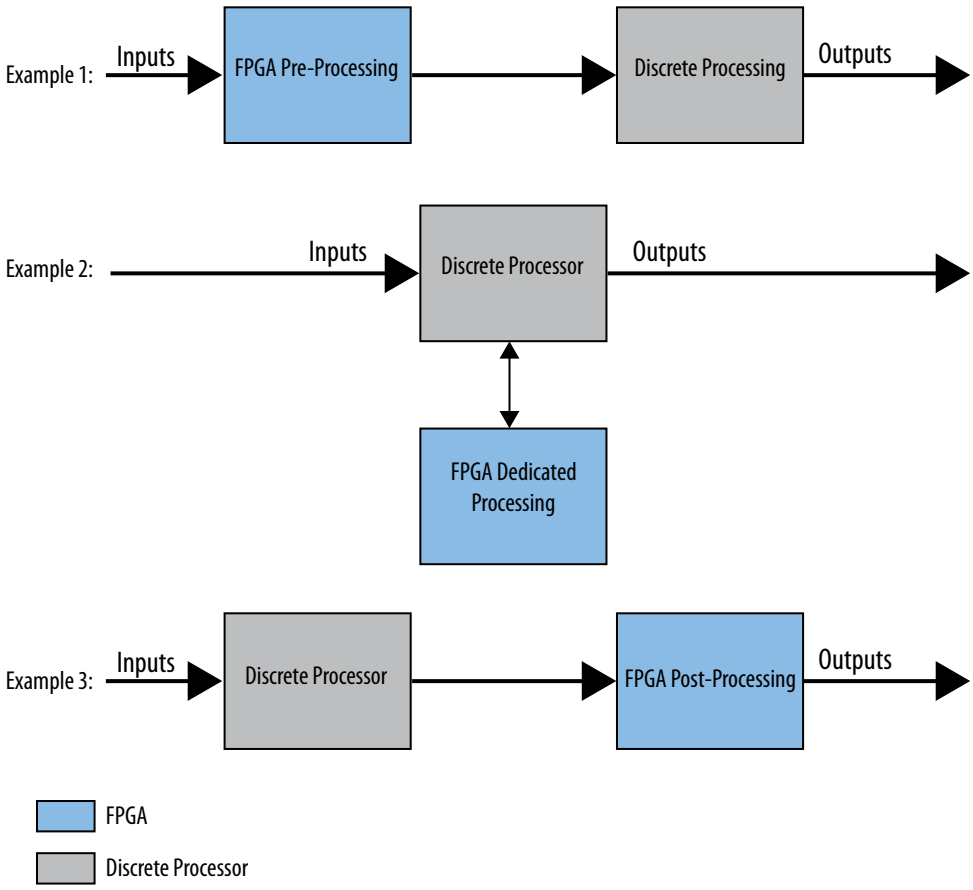
By off loading the processing required for the inputs or outputs to an FPGA, the discrete processor has more computation bandwidth available for the algorithmic processing.

Figure 7-10: Discrete Processing Stages



If the discrete processor requires more computational bandwidth for the algorithm, dedicated coprocessing can be added. The figure below shows examples of dedicated coprocessing at each stage.

Figure 7-11: Pre- Dedicated, and Post-Processing



Replacing State Machines

You can use the Nios II processor to implement scalable and efficient state machines. When you use dedicated hardware to implement state machines, each additional state or state transition increases the hardware utilization. In contrast, adding the same functionality to a state machine that runs on the Nios II processor only increases the memory utilization of the Nios II processor.

A key benefit of using Nios II for state machine implementation is the reduction of complexity that results from using software instead of hardware. A processor, by definition, is a state machine that contains many states. These states can be stored in either the processor register set or the memory available to the processor; consequently, state machines that would not fit in the footprint of a FPGA can be created using memory connected to the Nios II processor.

When designing state machines to run on the Nios II processor, you must understand the necessary throughput requirements of your system. Typically, a state machine is comprised of decisions (transitions) and actions (outputs) based on stimuli (inputs). The processor you have chosen determines the speed at which these operations take place. The state machine speed also depends on the complexity of the algorithm being implemented. You can subdivide the algorithm into separate state machines using software modularity or even multiple Nios II processor cores that interact together.

Low-Speed State Machines

Low-speed state machines are typically used to control peripherals. The Nios II/e processor pictured in **Figure 7-8** could implement a low speed state machine to control the peripherals.

Even though the Nios II/e core does not include a data cache, Altera recommends that the software accessing the peripherals use data cache bypassing. Doing so avoids potential cache coherency issues if the software is ever run on a Nios II/f core that includes a data cache.

For information about data cache bypass methods, refer to the Processor Architecture chapter of the *Nios II Gen2 Processor Reference Handbook*.

State machines implemented in Qsys require the following components:

- A Nios II processor
- Program and data memory
- Stimuli interfaces
- Output interfaces

The building blocks you use to construct a state machine in Qsys are no different than those you would use if you were creating a state machine manually. One noticeable difference in the Qsys environment is accessing the interfaces from the Nios II processor. The Nios II processor uses an Avalon-MM master port to access peripherals. Instead of accessing the interfaces using signals, you communicate via memory-mapped interfaces. Memory-mapped interfaces simplify the design of large state machines because managing memory is much simpler than creating numerous directly connected interfaces.

For more information about the Avalon-MM interface, refer to the Avalon Interface Specifications.

Related Information

- **Processor Architecture**
- **Avalon Interface Specifications**

High-Speed State Machines

You should implement high throughput state machine using a Nios II/f core. To maximize performance, focus on the I/O interfaces and memory types. The following recommendations on memory usage can maximize the throughput of your state machine:

- Use on-chip memory to store logic for high-speed decision making.
- Use tightly-coupled memory if the state machine must operate with deterministic latency. Tightly-coupled memory has the same access time as cache memory; consequently, you can avoid using cache memory and the cache coherency problems that might result.

For more information about tightly-coupled memory, refer to the Cache and Tightly-Coupled Memory chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

Cache and Tightly-Coupled Memory

Subdivided State Machines

Subdividing a hardware-based state machine into smaller more manageable units can be difficult. If you choose to keep some of the state machine functionality in a hardware implementation, you can use the Nios II processor to assist it. For example, you may wish to use a hardware state machine for the high data throughput functionality and Nios II for the slower operations. If you have partitioned a complicated state machine into smaller, hardware based state machines, you can use the Nios II processor for scheduling.

Software Application Optimization

This section examines techniques to increase your software application's performance and decrease its size.

Performance Tuning Background

Software performance is the speed with which a certain task or series of tasks can be performed in the system. To increase software performance, you must first determine the sections of the code in which the processing time is spent.

An application's tasks can be divided into interrupt tasks and system processing tasks. Interrupt task performance is the speed with which the processor completes an interrupt service routine to handle an external event or condition. System processing task performance is the speed with which the system performs a task explicitly described in the application code.

A complete analysis of application performance examines the performance of the system processing tasks and the interrupt tasks, as well as the footprint of the software image.

Speeding Up System Processing Tasks

To increase your application's performance, determine how you can speed up the system processing tasks it performs. First analyze the current performance and identify the slowest tasks in your system, then determine whether you can accelerate any part of your application by increasing processor efficiency, creating a hardware accelerator, or improving the applications's methods for data movement.

Analyzing the Problem

The first step to accelerate your system processing is to identify the slowest task in your system. Altera provides the following tools to profile your application:

- **GNU Profiler**—The Nios II EDS toolchain includes a method for profiling your application with the GNU Profiler. This method of profiling reports how long various functions run in your application.
- **High resolution timer**—The interval timer peripheral is a simple time counter that can determine the amount of time a given subroutine runs.
- **Performance counter peripheral**—The performance counter unit can profile several different sections of code with a collection of counters. This peripheral includes a simple software API that enables you to print out the results of these counters through the Nios II processor's stdio services.

Use one or more of these tools to determine the tasks in which your application is spending most of its processing time.

For more information about how to profile your software application, refer to *AN391: Profiling Nios II Systems*.

Related Information

AN391: Profiling Nios II Systems

Accelerating your Application

This section describes several techniques to accelerate your application. Because of the flexible nature of the FPGA, most of these techniques modify the system hardware to improve the processor's execution performance. This section describes the following performance enhancement methods:

- Methods to increase processor efficiency
- Methods to accelerate select software algorithms using hardware accelerators
- Using a DMA peripheral to increase the efficiency of sequential data movement operations

Increasing Processor Efficiency

An easy way to increase the software application's performance is to increase the rate at which the Nios II processor fetches and processes instructions, while decreasing the number of instructions the application requires. The following techniques can increase processor efficiency in running your application:

- **Processor clock frequency**—Modify the processor clock frequency using Qsys. The faster the execution speed of the processor, the more quickly it is able to process instructions.
- **Nios II processor improvements**—Select the most efficient version of the Nios II processor and parameterize it properly. The following processor settings can be modified using Qsys:
  - **Processor type**—Select the fastest Nios II processor core possible. In order of performance, from fastest to slowest, the processors are the Nios II/f and Nios II/e cores.
  - **Instruction and data cache**—Include an instruction or data cache, especially if the memory you select for code execution—where the application image and the data are stored—has high access time or latency.
  - **Multipliers**—Use hardware multipliers to increase the efficiency of relevant mathematical operations.

    For more information about the processor configuration options, refer to the Instantiating the Nios II Processor chapter of the *Nios II Gen2 Processor Reference Handbook*.
- **Nios II instruction and data memory speed**—Select memory with low access time and latency for the main program execution. The memory you select for main program execution impacts overall performance, especially if the Nios II caches are not enabled. The Nios II processor stalls while it fetches program instructions and data.
- **Tightly coupled memories**—Select a tightly coupled memory for the main program execution. A tightly coupled memory is a fast general purpose memory that is connected directly to the Nios II processor's instruction or data paths, or both, and bypasses any caches. Access to tightly coupled memory has the same speed as access to cache memory. A tightly coupled memory must guarantee a single-cycle access time. Therefore, it is usually implemented in an FPGA memory block.

  For more information about tightly coupled memories, refer to the *Using Tightly Coupled Memory with the Nios II Processor Tutorial* and to the Cache and Tightly-Coupled Memory chapter of the *Nios II Gen2 Software Developer's Handbook*.
- **Compiler Settings**—More efficient code execution can be attained with the use of compiler optimizations. Increase the compiler optimization setting to -03, the fastest compiler optimization setting, to attain more efficient code execution. You set the C-compiler optimization settings for the BSP project independently of the optimization settings for the application. For information about configuring the compiler optimization level for the BSP project, refer to the `hal.make.bsp_cflags_optimization` BSP setting in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Nios II Software Build Tools Reference**
- **Instantiating the Nios II Gen2 Processor**

- **Cache and Tightly-Coupled Memory**
- **Using Tightly Coupled Memory with the Nios II Processor Tutorial**

Accelerating Hardware

Slow software algorithms can be accelerated with the use of custom instructions, dedicated hardware accelerators. The following techniques can increase processor efficiency in running your application:

- Custom instructions—Use custom instructions to augment the Nios II processor's arithmetic and logic unit (ALU) with a block of dedicated, user-defined hardware to accelerate a task-specific, computational operation. This hardware accelerator is associated with a user-defined operation code, which the application software can call.

  For information about how to create a custom instruction, refer to *Nios II Custom Instruction User Guide*.
- Hardware accelerators—Use hardware accelerators for bulk processing operations that can be performed independently of the Nios II processor. Hardware accelerators are custom, user-defined peripherals designed to speed up the processing of a specific system task. They increase the efficiency of operations that are performed independently of the Nios II processor.

  For more information about hardware acceleration, refer to the Hardware Acceleration and Coprocessing chapter.

Related Information

- **Hardware Acceleration and Coprocessing** on page 7-1
- **Nios II Custom Instruction User Guide**

Improving Data Movement

If your application performs many sequential data movement operations, a DMA peripheral might increase the efficiency of these operations. Altera provides the following two DMA peripherals for your use:

- DMA—Simple DMA peripheral that can perform single operations before being serviced by the processor. For more information about using the DMA peripheral, refer to “HAL Peripheral Services” .

  For information about the DMA peripheral, refer to the DMA Controller Core chapter in the *Embedded Peripherals IP User Guide*.
- Scatter-Gather DMA (SGDMA)—Descriptor-based DMA peripheral that can perform multiple operations before being serviced by processor.

  For more information, refer to the Altera Modular Scatter-Gather DMA chapter in the *Embedded Peripherals IP User Guide*.

Related Information

- **Embedded Peripherals IP User Guide**
- **HAL Peripheral Services** on page 4-44

Accelerating Interrupt Service Routines

To increase the efficiency of your interrupt service routines, determine how you can speed up the tasks they perform. First analyze the current performance and identify the slowest parts of your interrupt dispatch and handler time, then determine whether you can accelerate any part of your interrupt handling.

Analyzing the Problem

The total amount of time consumed by an interrupt service routine is equal to the latency of the HAL interrupt dispatcher plus the interrupt handler running time. Use the following methods to profile your interrupt handling:

- Interrupt dispatch time—Calculate the interrupt handler entry time using the method found in design files that accompany the *Using Tightly Coupled Memory with the Nios II Processor Tutorial* on the Altera literature pages.You can download the design files from the Documentation: Tutorials page of the Altera website.
- Interrupt service routine time—Use a timer to measure the time from the entry to the exit point of the service routine.

Related Information

- **Using Tightly Coupled Memory with the Nios II Processor Tutorial**
- **Documentation: Tutorials**

Accelerating the Interrupt Service Routine

The following techniques can increase interrupt handling efficiency when running your application:

- **General software performance enhancements**—Apply the general techniques for improving your application's performance to the ISR and ISR handler. Place the .exception code section in a faster memory region, such as tightly coupled memory.
- **IRQ priority**—Assign an appropriate priority to the hardware interrupt. The method for assigning interrupt priority depends on the type of interrupt controller.
  - With the internal interrupt controller, set the interrupt priority of your hardware device to the lowest number available. The HAL ISR service routine uses a priority based system in which the lowest number interrupt has the highest priority.
  - With an external interrupt controller (EIC), the method for priority configuration depends on the hardware. Refer to the EIC documentation for details.
- **Custom instruction and tightly coupled memories**—Decrease the amount of time spent by the interrupt handler by using the interrupt-vector custom instruction and tightly coupled memory regions.
- **VIC block**—The VIC offers high-performance, low-latency interrupt handling. The VIC prioritizes interrupts in hardware and outputs information about the highest-priority pending interrupt. When external interrupts occur in a system containing a VIC, the VIC determines the highest priority interrupt, determines the source that is requesting service, computes the requested handler address (RHA), and provides information, including the RHA, to the processor. For more information, refer to the "Vectored Interrupt Controller Core" chapter of the *Embedded Peripheral IP User Guide*.

For more information about how to improve the performance of the Nios II exception handler, refer to the Exception Handling chapter of the *Nios II Gen2 Software Developer's Handbook*.

Related Information

- **Embedded Peripherals IP User Guide**
- **Exception Handling**

Reducing Code Size

Reducing the memory space required by your application image also enhances performance. This section describes how to measure and decrease your code footprint.

Analyzing the Problem

The easiest way to analyze your application's code footprint is to use the GNU Binary Utilities tool **nios2-elf-size**. This tool analyzes your compiled `.elf` binary file and reports the total size of your application, as well as the subtotals for the `.text`, `.data`, and `.bss` code sections. The example below shows a **nios2-elf-size** command response.

Example 7-1: Example Use of nios2-elf-size Command

```
> nios2-elf-size -d application.elf
text data bss dec hex filename
203412 8288 4936 216636 34e3c application.elf
```

Reducing the Code Footprint

The following methods help you to reduce your code footprint:

- **Compiler options**—Setting the `-Os` flag for the GCC causes the compiler to apply size optimizations for code size reduction. Use the `hal.make.bsp_cflags_optimization` BSP setting to set this flag.
- **Reducing the HAL footprint**—Use the HAL BSP library configuration settings to reduce the size of the HAL component of your BSP library file. However, enabling the size-reduction settings for the HAL BSP library often impacts the flexibility and performance of the system.

  The table below lists the configuration settings for size optimization. Use as many of these settings as possible with your system to reduce the size of BSP library file.

Table 7-1: BSP Settings to Reduce Library Size

BSP Setting Name	Value
hal.max_file_descriptors	4
hal.enable_small_c_library	True
hal.sys_clk_timer	None
hal.timestamp_timer	None
hal.enable_exit	False
hal.enable_c_plus_plus	False
hal.enable_lightweight_device_driver_api	True
hal.enable_clean_exit	False
hal.enable_sim_optimize	False
hal.enable_reduced_device_drivers	True
hal.make.bsp_cflags_optimization	\"-Os\"

  You can reduce the HAL footprint by adjusting BSP settings as shown in the table.
- **Removing unused HAL device drivers**—Configure the HAL with support only for system peripherals your application uses.
  - By default, the HAL configuration mechanism includes device driver support for all system peripherals present. If you do not plan on accessing all of these peripherals using the HAL device drivers, you can elect to have them omitted during configuration of the HAL BSP library by using the `set_driver` command when you configure the BSP project.
  - The HAL can be configured to include various software modules, such as the NicheStack networking stack and the read-only zip file system, whose presence increases the overall footprint of the application. However, the HAL does not enable these modules by default.

Memory Optimization

This section presents tips and tricks that can be helpful when implementing any type of memory in your Qsys system. These techniques can help improve system performance and efficiency.

Isolate Critical Memory Connections

For many systems, particularly complex ones, isolating performance-critical memory connections is beneficial. To achieve the maximum throughput potential from memory, connect it to the fewest number of masters possible and share those masters with the fewest number of slaves possible. Minimizing connections reduces the size of the data multiplexers required, increasing potential clock speed, and also reduces the amount of arbitration necessary to access the memory. You can use bridges to isolate memory connections.

Related Information

Avalon-MM Byte Ordering on page 3-19

Match Master and Slave Data Width

Matching the data widths of master and slave pairs in Qsys is advantageous. Whenever a master port is connected to a slave of a different data width, Qsys inserts adapter logic to translate between them. This logic can add additional latency to each transaction, reducing throughput. Whenever possible, try to keep the data width consistent for performance-critical master and slave connections. In cases where masters are connected to multiple slaves, and slaves are connected to multiple masters, it may be impossible to make all the master and slave connections the same data width. In these cases, you should concentrate on the master-to-slave connections which have the most impact on system performance.

For instance, if Nios II processor performance is critical to your overall system performance, and the processor is configured to run all its software from an SDRAM device, you should use a 32-bit SDRAM device because that is the native data width of the Nios II processor, and it delivers the best performance. Using a narrower or wider SDRAM device can negatively impact processor performance because of greater latency and lower throughput. However, if you are using a 64-bit DMA to move data to and from SDRAM, the overall system performance may be more dependent on DMA performance. In this case, it may be advantageous to implement a 64-bit SDRAM interface.

Use Separate Memories to Exploit Concurrency

When multiple masters in your system access the same memory, each master is granted access only some fraction of the time. Shared access may hurt system throughput if a master is starved for data.

If you create separate memory interfaces for each master, they can access memory concurrently at full speed, removing the memory bandwidth bottleneck. Separate interfaces are quite useful in systems which employ a DMA, or in multiprocessor systems in which the potential for parallelism is significant.

In Qsys, it is easy to create separate memory interfaces. Simply instantiate multiple on-chip memory components instead of one. You can also use this technique with external memory devices such as external SRAM and SDRAM by adding more, possibly smaller, memory devices to the board and connecting them to separate interfaces in Qsys. Adding more memory devices presents tradeoffs between board real estate, FPGA pins, and FPGA logic resources, but can certainly improve system throughput. Your system topology should reflect your system requirements.

Related Information

Avalon-MM Byte Ordering on page 3-19

Understand the Nios II Instruction Master Address Space

This Nios II processor instruction master cannot address more than a 256 MByte span of memory; consequently, providing more than 256 MBytes to run Nios II software wastes memory resources. This restriction does not apply to the Nios II data master, which can address as many as 2 GBytes.

Test Memory

You should rigorously test the memory in your system to ensure that it is physically connected and set up properly before relying on it in an actual application. The Nios II Embedded Design Suite ships with a memory test example which is a good starting point for building a thorough memory test for your system.

Accelerating Nios II Networking Applications

This section describes key optimizations you can use to accelerate the performance of your Nios II networking application. In addition, it describes how the different parts of a Nios II Ethernet-enabled system work together, how the interaction of these parts corresponds to the total networking performance of the system, and how to benchmark the system.

Ethernet is a standard data transport paradigm for embedded systems across all applications because it is inexpensive, abundant, mature, and reliable.

As seen in the empirical benchmark results, you can achieve minor performance increases in your Ethernet system by applying a single hardware optimization; however, achieving significant Ethernet performance increases involves applying several hardware optimizations together in the same system.

Consider using the following optimizations for your Ethernet system, in decreasing order of importance:

- Using a DMA engine for moving data to and from the Ethernet device
- Increasing the overall system frequency, including components such as the processor, DMA peripherals, and memory
- Using low-latency memory for Nios II software execution
- Using a custom hardware peripheral to accelerate the network checksum
- Using fast packet memory to store Ethernet data

Finally, the overall performance you can achieve from your Ethernet application depends on the nature of the application itself. This section provides you with general techniques to accelerate Nios II Ethernet applications, but the final measure of success is whether your application meets the performance goals you establish.

Downloading the Ethernet Acceleration Design Example

The Nios II ethernet acceleration design example is an integral part of this section. The design example shows how the acceleration techniques can be applied in a real working Nios II system. The **readme.doc** file, located in the design example folder, provides additional hands-on instructions that demonstrate how to implement the acceleration techniques in a Nios II system. The **readme.doc** file also provides performance benchmark results.

You can find the Nios II ethernet acceleration design example on the Nios II Ethernet Acceleration Design Example page of the Altera website.

Download the design example file, and unzip the file into a working directory.

Related Information

Nios II Ethernet Acceleration Design Example

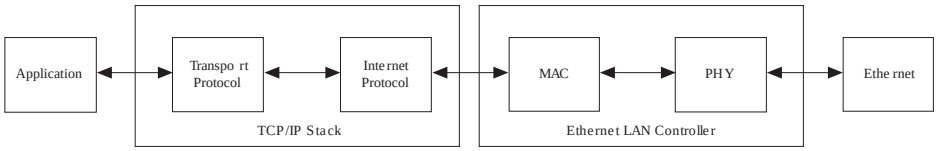
The Structure of Networking Applications

This section describes the different parts of a general networking application.

Ethernet System Hierarchy

The figure below shows the flow of information from an embedded networking application to the Ethernet.

Figure 7-12: The Ethernet System Hierarchy



The structure presented in the figure shows a typical embedded networking system. In general, a user application performs a job that defines the goal of the embedded system, such as controlling the speed of a motor or providing the UI for an embedded kiosk. The networking stack provides the application with an application programming interface (API), usually the Sockets API, to send networking data to and from the embedded system.

The stack itself is a software library that converts data from the user application into networking packets, and sends the packets through the networking device. Networking stacks tend to be very complicated software state machines that must be able to send data using a wide variety of networking protocols, such as address resolution protocol (ARP), transmission control protocol (TCP), and user datagram protocol (UDP). These stacks generally require a significant amount of processing power.

The stack uses the Ethernet device to move data across the physical media. Most of a networking stack's interaction with the networking device consists of shuttling Ethernet packets to and from the Ethernet device.

You must consider the link layer, or physical media over which the Ethernet datagrams travel, when constructing a network enabled system. Depending on the location of the embedded system, the Ethernet datagrams might traverse a wide variety of physical links, such as 10/100 Mb twisted pair and fiber optic. Additionally, the datagrams might experience latency if they traverse long distances or need to pass through many network switches in order to arrive at their destination.

Relationships Between Networking System Elements

The total throughput performance of an embedded networking system is highly dependent on the interaction of the user application, networking stack, Ethernet device (and driver), as well as the physical connection for the networking link. Making substantial performance improvements in the network throughput often depends on optimizing the performance of all these elements simultaneously.

In general, your networking application has some criteria for performance that are either achieved or not. However, a good first order approximation for determining the viability of your networking application is to remove the user application from the system and measure the total networking performance. This method provides you with an upper bound for total network performance, which you can use to create your networking application. This section uses a simple benchmark program that determines the raw throughput rate of TCP and UDP data transactions. This benchmark application does very little apart from sending or receiving data through the networking stack. It therefore provides us with a good approximation of the maximum networking performance achievable.

Finding the Performance Bottlenecks

A wide variety of tools are available for analyzing the performance of your Nios II embedded system and finding system bottlenecks. In this section, many of the techniques presented to increase overall system (and networking) performance were discovered through the use of the following tools:

- GNU profiler
- Timer peripheral IP core
- Performance counter IP core

This section does not explore the use of these tools or how they were applied to find networking bottlenecks in the system. For more information about finding general performance bottlenecks in your Nios II embedded system, refer to *AN:391 Profiling Nios II Systems*.

Related Information

AN:391 Profiling Nios II Systems

The User Application

In an embedded networking system, the application layer is the part of the system where your key task is performed. In general, this application layer performs some work and then uses the network stack to send and receive data. In a classic embedded networking system, your application executes on the same processor as the network stack, and competes with it for computation resources.

To increase the throughput of your networking system, decrease the time your application spends completing its task between the function calls it makes to the networking stack. This technique has a twofold benefit. First, the faster your application runs to completion before sending or receiving data, the more function calls it can make to the networking stack (Sockets API) to move data across the network. Second, if the application takes less of the processor's time to run, the more time the processor has to operate the networking stack (and networking device) and transmit the data.

User Application Optimizations

This section describes some effective ways to decrease the amount of time your application uses the Nios II processor.

Software Optimizations

- Compiler Optimization Level—Compile your application with the highest compiler optimization possible. Higher optimizations result in denser, faster code, increasing the computational efficiency of the processor.
- MicroC/OS-II Thread Priority—Make sure that your application task has the right MicroC/OS-II priority level assigned to it. In general, the higher the priority of the application, the faster it runs to completion. Balance the application's priority levels against the priority levels assigned to the NicheStack's core tasks, discussed in "Structure of the NicheStack Networking Stack".

Note: This suggestion assumes that your application uses Altera's recommended method for operating the NicheStack Networking Stack, which requires using the MicroC/OS-II operating system.

Related Information

Structure of the NicheStack Networking Stack on page 7-28

Hardware Optimizations

- Processor Performance—You can increase the performance of the Nios II processor in the following ways:
  - Computational Efficiency—Selecting the most computationally efficient Nios II processor core is the quickest way to improve overall application performance. The following Nios II processor cores are available, in decreasing order of performance:
    - Nios II/f—optimized for speed
    - Nios II/e—conserves on-chip resources at the expense of speed
  - Memory Bandwidth—Using low-latency, high speed memory decreases the amount of time required by the processor to fetch instructions and move data. Additionally, increasing the processor's arbitration share of the memory increases the processor's performance by allowing the Nios II processor to perform more transactions to the memory before another Avalon master port can assume control of the memory.
  - Instruction and Data Caches—Adding an instruction and data cache is an effective way to decrease the amount of time the Nios II processor spends performing operations, especially in systems that have slow memories, such as SDRAM or double data rate (DDR) SDRAM. In general, the larger the cache size selected for the Nios II processor, the greater the performance improvement.
  - Clock Frequency—Increasing the speed of the processor's clock results in more instructions being executed per unit of time. To gain the best performance possible, ensure that the processor's execution memory is in the same clock domain as the processor, to avoid the use of clock-crossing adapters.

    One of the easiest ways to increase the operational clock frequency of the processor and memory peripherals is to use a pipeline bridge IP core to isolate the slower peripherals of the system. With this peripheral, the processor, memory, and Ethernet device are connected on one side of the bridge. On the other side of the bridge are all of the peripherals that are not performance dependent.
- Hardware Acceleration—Hardware acceleration can provide tremendous performance gains by moving time-intensive processor tasks to dedicated hardware blocks in the system. The following list contains most common ways to accelerate application level algorithms:
  - Custom Instruction—Offload the Nios II processor by using hardware to implement a custom instruction.
  - Custom Peripheral—Create a block of hardware that performs a specific algorithmic task, as a peripheral controlled by the Nios II processor.

Related Information

Hardware Acceleration and Coprocessing on page 7-1

The Sockets API

After tuning your application to become more computationally efficient (thereby freeing more of the processor's time for operating the networking stack), you can optimize how the application uses the networking stack. This section describes how to select the best protocol for use by your application and the most efficient way to use the Sockets API.

Selecting the Right Networking Protocol

When using the Sockets API, you must also select which protocol to use for transporting data across the network. There are two main protocols used to transport data across networks: TCP and UDP. Both of these protocols perform the basic function of moving data across Ethernet networks, but they have very different implementations and performance implications. The table below compares the two protocols.

Table 7-2: The UDP and TCP Protocols

Parameter	Protocol	
UDP	TCP	
Connection Mode	Connectionless	Connection-Oriented
In Order Data Guarantee	No	Yes
Data Integrity and Validation	No	Yes
Data Retransmission	No	Yes
Data Checksum	Yes; Can be disabled	Yes

In terms of just throughput performance, the UDP protocol is much faster than TCP because it has very little overhead. The UDP protocol makes no attempt to validate that the data being sent arrived at its destination (or even that the destination is capable of receiving packets), so the network stack needs to perform much less work in order to send or receive data using this protocol.

However, aside from very specialized cases where your embedded system can tolerate losing data (for example, streaming multimedia applications), use the TCP protocol.

Note: Use the UDP protocol to gain the fastest performance possible; however, use the TCP protocol when you must guarantee the transmission of the data.

Improving Send and Receive Performance

Proper use of the Sockets API in your application can also increase the overall networking throughput of your system. The following list describes several ways to optimally use the Sockets API:

- Minimize send and receive function calls—The Sockets API provides two sets of functions for sending and receiving data through the networking stack. For the UDP protocol these functions are `sendto()` and `recvfrom()`. For the TCP protocol these functions are `send()` and `recv()`.

  Depending on which transport protocol you use (TCP or UDP), your application uses one of these sets of functions. To increase overall performance, avoid calling these functions repetitively to handle small units of data. Every call to these functions incurs a fixed time penalty for execution, which can compound quickly when these functions are called multiple times in rapid succession. Combine data that you want to send (or receive) and call these functions with the largest possible amount of data at one time.

  Note: Call the Socket API's send and receive functions with larger buffer sizes to minimize system call overhead.
- Minimize latency when sending data—Although the TCP Sockets `send()` function can accept an arbitrary number of bytes, those bytes might not be immediately sent as a packet. This situation is especially likely when `send()` is called with a small number of bytes, because the networking stack attempts to coalesce these small data chunks into a larger packet. Small data chunks are coalesced to avoid congesting the network with many small packets (using the Nagle algorithm for congestion avoidance). There is a solution, however, through the use of the `TCP_NODELAY` flag.

  Setting a socket's `TCP_NODELAY` flag, with the `setsockopt()` function call, disables the Nagle algorithm. The socket immediately sends whatever bytes are passed in as a TCP packet. Disabling the Nagle algorithm can be a useful way to increase network throughput in the case where your application must send many small chunks of data very quickly.

  Note: If you need to accelerate the transmission of small TCP packets, use the `TCP_NODELAY` flag on your socket. You can find an example of setting the `TCP_NODELAY` flag in the benchmarking application software in the Nios II ethernet acceleration design example.

  While disabling the Nagle algorithm usually causes smaller packets to be immediately sent over the network, the networking stack might still coalesce some of the packets into larger packets. This situation is especially likely in the case of the Windows workstation platform. However, you can expect the networking stack to do so with much lower frequency than if the Nagle algorithm were enabled.

The Zero Copy API

The NicheStack networking stack provides a further optimization to accelerate the data transfers to and from the stack called the zero copy API. The zero copy API increases overall system performance by eliminating the buffer management scheme performed by the Socket API's read and write function calls. The application manages the send and receive data buffers directly, eliminating an extra level of data copying performed by the Nios II processor.

Using the NicheStack Zero Copy API can accelerate your network application's throughput by eliminating an extra layer of copying.

Structure of the NicheStack Networking Stack

The NicheStack networking stack is a highly-configurable software library designed for communicating over TCP/IP networks. The version that Altera ships in the Nios II Embedded Design Suite (EDS) is optimized for use with the MicroC/OS-II (RTOS), and includes device driver support for the Altera Triple Speed Ethernet MegaCore function, which serves as the media access control (MAC).

The NicheStack networking stack is extremely configurable, with the entire software library utilizing a single configuration header file, called **ipport.h**.

General Optimizations

Because this section focuses on a single Nios II system, most of the optimizations described in "User Application Optimizations" also improve the performance of the NicheStack networking stack. The following optimizations also help increase your overall network performance:

- Software optimizations
  - Compiler optimization level
- Hardware optimizations
  - Processor performance
    - Computational efficiency
    - Memory bandwidth
    - Instruction and data caches
    - Clock frequency

Related Information

User Application Optimizations on page 7-25

NicheStack Specific Optimizations

This section describes the targeted optimizations that you can use to increase the performance of the NicheStack networking stack directly.

NicheStack Thread Priorities

Altera's version of the NicheStack networking stack relies on the MicroC/OS-II operating system's threads to drive two critical tasks to properly service the networking stack. These tasks (threads) are **tk_nettick**, which is responsible for timekeeping, and **tk_netmain**, which is used to drive the main operation of the stack.

When building a NicheStack-based system in the Nios II EDS, the default run-time thread priorities assigned to these tasks are: **tk_netmain** = 2 and **tk_nettick** = 3. These thread priorities provide the best networking performance possible for your system. However, in your embedded system you might need to override these priorities because your application task (or tasks) run more frequently than these tasks. Overriding these priorities, however, might result in performance degradation of network operations, as the NicheStack networking stack has fewer processor cycles to complete its tasks.

Therefore, if you need to increase the priority of your application tasks above that of the NicheStack tasks, make sure to yield control whenever possible to ensure that these tasks get some processor time. Additionally, ensure that the **tk_netmain** and **tk_nettick** tasks have priority levels that are just slightly less than the priority level of your critical system tasks.

When you yield control, the MicroC/OS-II scheduler places your application task from a running state into a waiting state. The scheduler then takes the next ready task and places it into a running state. If tk_netmain and tk_nettick are the higher priority tasks, they are allowed to run more frequently, which in turn increases the overall performance of the networking stack.

Note: If your MicroC/OS-II based application tasks run with a higher priority level (lower priority number) than the NicheStack tasks, remember to yield control periodically so the NicheStack tasks can run. Tasks using the NicheStack services should call the function `tk_yield()`. If they do not use the NicheStack services, the tasks should call the function `OSTimeDly()`.

Disabling Nonessential NicheStack Modules

Because the NicheStack networking stack is highly configurable, many modules are available for you to optionally include. Some examples are an FTP client, an FTP server, and a web server. Every module included in your system might result in some performance degradation due to the overhead associated with having the Nios II processor service these modules.

This degradation can happen because the main NicheStack task, **tk_netmain**, periodically polls each of these modules. Also, these modules might insert time-based callback functions, which further decrease the overall performance of the networking stack.

You can control what is enabled or disabled in the NicheStack networking stack through a series of macro definitions in the **ipport.h** configuration file. In addition, the NicheStack software component inserts some definitions in the **system.h** file belonging to the board support package (BSP). A list of NicheStack features and modules to disable, which can increase system performance, follows. (To disable a particular feature or module, ensure that its `#define` statement is present in neither the **ipport.h** file nor the **system.h** configuration file.)

The NicheStack features to disable include the following items:

- IN_MENUS—enable NicheTool command interface
- NPDEBUG—enable debugging aids
- MEM_WRAPPERS—debugging aid to validate memory
- QUEUE_CHECKING—debugging aid to validate memory queues
- MULTI_HOMED—not needed if only one networking device
- IP_ROUTING—not needed if only one networking device

The NicheStack modules to disable include the following items:

- PING_APP—enable ping support
- UDPSTEST, TCP_ECHOTEST—enable echotest programs
- FTP CLIENT, FTP SERVER—enable FTP client/server
- TELNET_SVR—enable Telnet server
- USE_SYSLOG_TASK—enable statistics collection
- SMTP_ALERTS—enable email client
- INCLUDE_SNMP—enable simple network management protocol (SNMP) server
- DNS_SERVER—enable domain name system (DNS) server

Disabling unused NicheStack networking stack features and modules in your system helps increase overall system performance.

The NicheStack networking stack also supports a wide variety of features and modules not listed here. Refer to the NicheStack documentation and your **ipport.h** file for more information.

Using Faster Packet Memory

You can increase the performance of the NicheStack networking stack by using fast, low-latency memory for storing Ethernet packets. This section describes this optimization and explains how it works.

Background

The NicheStack networking stack uses a memory queue to assemble and receive network packets. To send a packet, the NicheStack removes a free memory buffer from the queue, assembles the packet data into it, and passes this buffer memory location to the Ethernet device driver. To receive the data, the Ethernet device driver removes a free memory buffer, loads it with the received packet, and passes it back to the networking stack for processing. The NicheStack networking stack allows you to specify where its queue of buffer memory is located and how this memory allocation is implemented.

By default, the Altera version of the NicheStack networking stack allocates this pool of buffer memory using a series of `calloc()` function calls that use the system's heap memory. Depending on the design of the system, and where the Nios II system memory is located, this allocation method could impact overall system performance. For example, if your Nios II processor's heap segment is in high latency or slow memory, this allocation method might degrade performance.

Additionally, in the case where the Ethernet device utilizes direct memory access (DMA) hardware to move the packets and the Nios II processor is not directly involved in transmitting or receiving the packet data, then this buffer memory must exist in an uncached region. Lack of buffer caching further degrades the performance because the Nios II processor's data cache is not able to offset any performance issues due to the slow memory.

The solution is to use the fastest memory possible for the networking stacks buffer memory, preferably a separate memory not used by the Nios II processor for programmatic execution.

Solution

The ipport.h file defines a series of macros for allocating and deallocating big and small networking buffers. The macro names begin with `BB_` (for "big buffer") and `LB_` (for "little buffer"). Following is the block of macros with the definitions in place for Triple Speed Ethernet device driver support.

```
#define BB_ALLOC(size) ncpalloc(size)
#define BB_FREE(ptr) ncpfree(ptr)
#define LB_ALLOC(size) ncpalloc(size)
#define LB_FREE(ptr) ncpfree(ptr)
```

You can use these macros to allocate and deallocate memory any way you choose. The Nios II ethernet acceleration design example redefines these macros to allocate memory from on-chip memory (a fast memory structure inside the FPGA). This faster memory results in various degrees of performance increase, depending on the system. For detailed performance improvement figures, please refer to the **readme.doc** file included in the design example.

The Altera version of NicheStack does not use the `BB_FREE()` or `LB_FREE()` function calls. Therefore, any memory allocated with the `BB_ALLOC()` and `LB_ALLOC()` function calls is allocated at run time, and is never freed.

Using fast, low-latency memory for NicheStack's packet storage can improve the overall performance of the system.

Super Loop Mode

Although the Altera-supported version of the NicheStack networking stack requires MicroC/OS-II for its operation, you can configure the stack to run without an operating system. In this mode of operation, MicroC/OS-II is replaced by an infinite loop that services the stack and runs the user application.

Removing the MicroC/OS-II operating system from your system can result in slightly higher networking performance, but this improvement comes at the expense of additional complexity in the software design of your system. It is very easy to create pathological systems where your application code consumes all of the processor's time, and without frequent calls to a stack servicing function, the effective networking performance deteriorates.

Although the super loop system is another possible method of optimization, this section does not attempt to benchmark it.

You can use the NicheStack networking stack without the MicroC/OS-II operating system. Doing so can provide additional networking performance benefits. However, Altera does not support this configuration.

Ethernet Device

An important parameter in the total performance of your Ethernet application is the function and capabilities of the network interface device itself. The function of this device is to translate the physical Ethernet packets into datagrams that can be accessed by the stack. Therefore, its performance is critical to the overall performance of your networking application.

Link Speed

For most embedded networking applications, the network physical layer consists of either 100BASE-TX or 1000BASE-T Ethernet, which uses twisted copper wires for the transport medium. The maximum data transport rate (in one direction) for 100BASE-TX is 100 Mbps, while 1000BASE-T can accommodate 1000 Mbps.

It is very difficult for an embedded networking device to completely use a 100 Mb link, much less a 1000 Mb link. However, a faster link provides better performance most of the time, because the 1000 Mb link has a larger overall carrying capacity for data. The improvement is especially noticeable in cases where several different devices share the link and use it simultaneously.

Network Interface (Altera Triple Speed Ethernet MegaCore Function)

The Nios II EDS supports the Altera Triple Speed Ethernet MegaCore function. The Triple Speed Ethernet MegaCore function's role is essentially to translate an application's Ethernet data into physical bits on the Ethernet link. The Triple Speed Ethernet MegaCore function supports 10/100/1000 Mb networks. The table below lists the key design parameters that impact network performance.

Table 7-3: Triple Speed Ethernet MegaCore Function

Parameter	Altera Triple Speed Ethernet MegaCore Function
Type	FPGA IP
Control Interface	Avalon-MM
Data Interface	Avalon-ST
Data Width (bits)	8, 32
Supported Link Speeds (Mbps)	10/100/1000
Recv FIFO Depth	64 to 65536 entries
Send FIFO Depth	64 to 65536 entries
DMA	Altera Scatter-Gather DMA (required)
PHY Interface (Integrated)	None
PHY Interface (External)	MII (100 Mbps) GMII (1000 Mbps) RGMII(1000 Mbps) SGMII (10/100/1000 Mbps)

The Triple Speed Ethernet MegaCore function is capable of sending and receiving Ethernet data quickly because of the Scatter-Gather DMA peripherals. The Triple Speed Ethernet MegaCore function also allows you to select from a flexible range of send and receive FIFO depths.

NicheStack Device Driver Model

The NicheStack networking stack presents a simplified device driver model for integrating Ethernet devices, and the Altera Triple Speed Ethernet MegaCore function solution is fully optimized to support this model.

In the Triple Speed Ethernet MegaCore function device driver, the Scatter-Gather DMA peripherals are responsible for the movement of the Ethernet packet data to and from the Triple Speed Ethernet MegaCore function.

The Scatter-Gather DMA peripherals can operate much more efficiently than the Nios II processor for data movement operations (on a per clock basis), and therefore using the Triple Speed Ethernet MegaCore function device driver results in an overall performance increase in the system.

For information about the Triple Speed Ethernet MegaCore function, refer to the Triple Speed Ethernet MegaCore Function User Guide. For information about the Scatter-Gather DMA peripheral, refer to the *Embedded Peripherals IP User Guide*.

Related Information

- **Triple Speed Ethernet MegaCore Function User Guide**
- **Scatter-Gather DMA Controller Core**

Benchmarking Setup, Results, and Analysis

The previous sections describe several optimizations that you can use to increase the performance of a networking system. This section describes a method to evaluate the effectiveness of each one. The best way to evaluate the optimizations is to use a benchmarking application that measures the impact of applying each optimization.

Overview

A simple benchmarking application measures the overall networking performance. This application enables you to measure the Ethernet data transfer rate between two systems, such as an Altera development board and a workstation using the TCP or UDP protocols.

During a benchmarking test, one machine assumes the role of the sender and the other machine becomes the receiver. The sender opens a connection to the receiver, transmits a specified amount of data, and prints out a throughput measurement in Mbps. Likewise, the receiver waits for a connection from the sender, begins receiving Ethernet data, and, at the end of the data transmission, prints out the total throughput in Mbps.

The benchmarking application has the simplest possible structure. Both the sender and receiver parts of the program perform no additional work apart from sending and receiving Ethernet data. Additionally, for standardization purposes, all network operations use the industry standard Sockets API in their implementation.

Note: For more information about the benchmarking program, including detailed information about how to build and operate it, refer to the **readme.doc** file in the Nios II ethernet acceleration design example.

Test Setup

The benchmarking tests were conducted between a workstation and an Altera development board. The Altera development board used was a Stratix® IV GX development board. The workstation was lightly loaded, meaning that the only user applications running were the benchmark program and the Nios II Software Build Tools (SBT) for Eclipse.

The direct Ethernet connection between the two systems was implemented using a single twisted-pair networking cable.

Test Systems

The benchmarking analysis demonstrates how changing key parameters in an Ethernet system can lead to radical performance changes.

This benchmark test examines the merits of applying various optimizations to both the Nios II processor and the NicheStack networking stack. The first parameter tested is the effect of doubling the instruction and data cache sizes for the processor. The second parameter tested is the effect of increasing the Nios II processor's clock frequency.

The test also measures the effect of using fast internal memory for packet storage.

Test Methodology

This section describes the parameters used in the benchmarking tests.

Ethernet Link Type

The Ethernet link selected to connect the workstation to the Nios II board uses a single 100/1000 Mb cable in a point-to-point configuration (no hub or switch). This choice mitigates the potential effects of an additional piece of networking hardware on the test system.

In most networking applications, however, your system can be connected to another host through one (or more) Ethernet hubs or switches. These extra connections can increase the communication latency. The benchmark numbers present the idealized performance of an almost perfect Ethernet connection.

Protocols Tested

All benchmark operations are conducted using the TCP protocol. The TCP protocol guarantees that all data sent by the transmitter arrives at the receiver, ensuring that the throughput numbers reported are legitimate.

The benchmark application can measure UDP transmission speeds, but does so without accounting for lost or missing Ethernet packets. Therefore, the UDP test only measures the speed at which the transmitter can send all of the data using the UDP protocol, without considering whether the data arrived at the receiver.

Data Transmission Sizes

This series of tests uses a total data size of 100 megabytes (100,000,000 bytes). This data size increases the total amount of time spent in the course of the test, to more clearly capture the average performance of both the sender and receiver.

Furthermore, the tests use the largest TCP payload size for Ethernet packet transmission (1458 bytes). This payload size provides an upper bound of Ethernet performance, representing the best expected performance numbers achievable in the design.

Note: Because the benchmarking application uses the Sockets API, the payload size (1458 bytes) directly maps to the length parameter in the `send()` (TCP) and `sendto()` (UDP) function calls. The following statement is an example of a `send()` function call in TCP:

```
send(int <socket>, const void *<buffer>, size_t <length>, int <flags>);
```

Test Runs

For every Nios II configuration, the test measures the data transmission time and average data throughput with the Nios II system as both the sender and the receiver. The tests take three consecutive measurements and record the average of these runs as the final measurement.

Nios II System Software Configuration

For every Nios II configuration, the test measures the data transmission time and average data throughput with the Nios II system as both the sender and the receiver. The tests take three consecutive measurements and record the average of these runs as the final measurement.

NicheStack Networking Stack Configuration

The NicheStack networking stack is built with the default configuration. This configuration provides a minimal set of general purpose functionality to enabled networking operations using the TCP and UDP protocols.

Additionally, the following MicroC/OS-II thread priorities were selected for the two core NicheStack tasks:

- tk_netmain = priority 2
- tk_nettick = priority 3

MicroC/OS-II Configuration

The default MicroC/OS-II configuration is used for the operation of the networking stack. This configuration provides all the basic MicroC/OS-II services.

Benchmark Application

The benchmark application uses the Sockets API. The following configuration is used for the application:

- benchmark application = priority 4
- benchmark initialization thread = priority 1

Note: For more information about the benchmark application and its operation, refer to the Nios II ethernet acceleration design example.

General Application and System Library Settings

Both the benchmark application and the associated system library were compiled using the Nios II GNU tool chain with the `-O3` optimization enabled. If the test cases involve any changes to the run-time memory, the entire memory would be selected for the application's binary segments, such as `.text`, `.data`, and `.bss`.

Workstation System Software

The workstation benchmark application is compiled using the GNU tool chain for the Cygwin environment, targeting the x86 architecture. Because the workstation benchmark application reuses much of the same source code base as the Nios II application, it uses the Sockets API for conducting this test.

Nios II Test Hardware and Test Results

For details regarding the Nios II test hardware and test results, refer to the **readme.doc** file included in the Nios II ethernet acceleration design example.

Document Revision History

Table 7-4: Optimizing Nios II Based Systems and Software Chapter Revision History

Date	Version	Changes
December 2016	2016.12.19	Initial release.