

Embedded Design Handbook

Subscribe
Send Feedback

EDH | 2020.07.22
Latest document on the web: PDF | HTML

Contents

1. Introduction..... 6
1.1. Document Revision History for Embedded Design Handbook..... 6
2. First Time Designer's Guide..... 8
2.1. FPGAs and Soft-Core Processors..... 8
2.2. Embedded System Design..... 9
2.3. Embedded Design Resources..... 11
2.3.1. Intel Embedded Support..... 11
2.3.2. Intel Embedded Training..... 11
2.3.3. Intel Embedded Documentation..... 12
2.3.4. Third Party Intellectual Property..... 12
2.4. Intel Embedded Glossary..... 13
2.5. First Time Designer's Guide Revision History..... 14
3. Hardware System Design with Intel Quartus Prime and Platform Designer..... 15
3.1. FPGA Hardware Design..... 16
3.1.1. Connecting Your FPGA Design to Your Hardware..... 17
3.1.2. Connecting Signals to your Platform Designer System..... 17
3.1.3. Constraining Your FPGA-Based Design..... 18
3.2. System Design with Platform Designer..... 19
3.2.1. Intel System on a Programmable Chip (Platform Designer) Solutions..... 20
3.2.2. Platform Designer Design..... 22
3.3. Interfacing an External Processor to an Intel FPGA..... 23
3.3.1. Configuration Options..... 24
3.3.2. RapidIO Interface..... 27
3.3.3. PCI Express Interface..... 29
3.3.4. PCI Interface..... 31
3.3.5. Serial Protocol Interface (SPI)..... 31
3.3.6. Custom Bridge Interfaces..... 33
3.4. Avalon-MM Byte Ordering..... 35
3.4.1. Endianness..... 35
3.4.2. Avalon-MM Interface Ordering..... 36
3.4.3. Nios II Processor Data Accesses..... 40
3.4.4. Adapting Processor Masters to be Avalon-MM Compliant..... 42
3.4.5. System-Wide Design Recommendations..... 50
3.5. Memory System Design..... 52
3.5.1. Memory Types..... 52
3.5.2. On-Chip Memory..... 52
3.5.3. External SRAM..... 55
3.5.4. Flash Memory..... 56
3.5.5. SDRAM..... 58
3.5.6. Case Study..... 64
3.6. Nios II Hardware Development Tutorial..... 68
3.6.1. Software and Hardware Requirements..... 68
3.6.2. Intel FPGA IP Evaluation Mode..... 69
3.6.3. Nios II Design Example..... 69
3.6.4. Nios II System Development Flow..... 71
3.6.5. Creating the Design Example..... 75

3.7. Platform Designer System Design Tutorial..... 89
3.7.1. Software and Hardware Requirements..... 90
3.7.2. Download and Install the Tutorial Design Files.....91
3.7.3. Open the Tutorial Project.....91
3.7.4. Creating Platform Designer Systems.....92
3.7.5. Assemble a Hierarchical System..... 100
3.7.6. Viewing the Memory Tester System in Platform Designer..... 107
3.7.7. Compiling and Downloading Software to a Development Board..... 107
3.7.8. Debugging Your Design..... 109
3.7.9. Verifying Hardware in System Console..... 109
3.7.10. Simulating Custom Components..... 111
3.7.11. View a Diagram of the Completed System.....117
3.8. Hardware System Design with Intel Quartus Prime and Platform Designer Revision History.....118

4. Software System Design with a Nios II Processor 119
4.1. Nios II Command-Line Tools..... 120
4.1.1. Intel Command-Line Tools for Board Bringup and Diagnostics.....120
4.1.2. Intel Command-Line Tools for Flash Programming.....122
4.1.3. Intel Command-Line Tools for Software Development and Debug..... 125
4.1.4. Intel Command-Line Nios II Software Build Tools.....128
4.1.5. Rebuilding Software from the Command Line..... 129
4.1.6. GNU Command-Line Tools..... 130
4.2. Developing Nios II Software..... 137
4.2.1. Software Development Cycle..... 138
4.2.2. Software Project Mechanics..... 142
4.2.3. Developing With the Hardware Abstraction Layer.....161
4.2.4. Linking Applications.....182
4.3. Nios II MPU Usage.....184
4.3.1. Requirements..... 184
4.3.2. General Usage..... 184
4.3.3. Nios II MPU Design Examples.....192
4.4. Profiling Nios II Systems..... 201
4.4.1. Requirements..... 201
4.4.2. Tools..... 201
4.4.3. Using the GNU Profiler to Measure Code Performance.....203
4.4.4. Using Performance Counter and Timer Components..... 210
4.4.5. Troubleshooting..... 216
4.5. Software System Design with a Nios II Processor Revision History..... 217

5. Nios II Configuration and Booting Solutions..... 219
5.1. Introduction..... 219
5.1.1. Prerequisites..... 219
5.2. Nios II Processor Booting Methods.....220
5.2.1. Introduction to Nios II Booting Methods..... 221
5.2.2. Nios II Processor Booting from On-Chip Flash (UFM)..... 226
5.2.3. Nios II Processor Booting from EPCQ Flash..... 254
5.2.4. Nios II Processor Booting from QSPI Flash.....275
5.2.5. Nios II Processor Booting from On-Chip Memory (OCRAM).....306
5.2.6. Nios II Processor Booting from CFI Flash..... 314
5.2.7. Summary of Nios II Processor Vector Configurations and BSP Settings..... 349
5.3. Alternative Nios II Boot Methods..... 352

5.3.1. Assumptions About the Reader....352
5.3.2. Implementing a Custom Boot Copier....352
5.3.3. Default Nios II Boot Copier....353
5.3.4. Advanced Boot Copier Example....355
5.3.5. Implementing the Advanced Boot Copier Example....361
5.3.6. Small Boot Copier Example....369
5.3.7. Implementing the Small Boot Copier Example....370
5.3.8. Debugging Boot Copiers....375
5.3.9. Externally Controlling the Nios II Boot Process....376
5.4. Boot Time Performance Analysis....382
5.4.1. Boot Time Performance Analysis Design Example....382
5.4.2. Boot Time Measurement Strategy....383
5.4.3. Reducing Nios II Boot Time in Intel MAX 10 FPGA Design....384
5.4.4. Boot Time Performance and Estimation....387
5.5. Nios II Configuration and Booting Solutions Revision History....391

6. Nios II Debug, Verification, and Simulation....392
6.1. Software Debugging Options....393
6.2. Debugging Nios II Designs....395
6.2.1. Debuggers....395
6.2.2. Run-Time Analysis Debug Techniques....404
6.2.3. Using the Debug Code from Intel MAX 10 On-Chip Flash - User Flash Memory (UFM)....409
6.3. Verification and Board Bring-Up....418
6.3.1. Verification Methods....418
6.3.2. Board Bring-up....423
6.3.3. System Verification....429
6.4. Additional Embedded Design Considerations....433
6.4.1. JTAG Signal Integrity....433
6.4.2. Memory Space For System Prototyping....433
6.5. Simulating Nios II Embedded Processor Designs....434
6.5.1. Before You Begin....434
6.5.2. Setting Up and Generating Your Simulation Environment in Platform Designer 435
6.5.3. Creating the Nios II Software....436
6.5.4. Running Simulation in the ModelSim Simulator Using Nios II SBT for Eclipse.. 438
6.5.5. Running Simulation in the ModelSim Simulator Using Command Line....438
6.6. Nios II Debug, Verification, and Simulation Revision History....440

7. Optimizing Nios II Based Systems and Software....441
7.1. Hardware Acceleration and Coprocessing....442
7.1.1. Hardware Acceleration....442
7.1.2. Coprocessing....451
7.2. Software Application Optimization....459
7.2.1. Performance Tuning Background....459
7.2.2. Speeding Up System Processing Tasks....459
7.2.3. Accelerating Interrupt Service Routines....463
7.2.4. Reducing Code Size....464
7.3. Memory Optimization....466
7.3.1. Isolate Critical Memory Connections....466
7.3.2. Match Master and Slave Data Width....466
7.3.3. Use Separate Memories to Exploit Concurrency....466
7.3.4. Understand the Nios II Instruction Master Address Space....467

7.3.5. Test Memory....467
7.4. Accelerating Nios II Networking Applications....468
7.4.1. Downloading the Ethernet Acceleration Design Example....468
7.4.2. The Structure of Networking Applications....468
7.4.3. The User Application....470
7.4.4. Structure of the NicheStack Networking Stack....474
7.4.5. Ethernet Device....477
7.4.6. Benchmarking Setup, Results, and Analysis....479
7.4.7. Nios II Test Hardware and Test Results....482
7.5. Using Tightly Coupled Memory with the Nios II Processor Tutorial....483
7.5.1. Reasons for Using Tightly Coupled Memory....483
7.5.2. Tradeoffs....484
7.5.3. Guidelines for Using Tightly Coupled Memory....484
7.5.4. Tightly Coupled Memory Interface....486
7.5.5. Building a Nios II System with Tightly Coupled Memory....487
7.5.6. Generate the Platform Designer System....492
7.5.7. Run the Tightly Coupled Memories Examples from the Nios II Command....492
7.5.8. Program and Run the Tightly Coupled Memory Project....494
7.5.9. Understanding the Tcl Scripts....495
7.6. Optimizing Nios II Based Systems and Software Revision History....499

1. Introduction

The Embedded Design Handbook complements the primary documentation for the Intel 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 Intel-provided tools. The handbook introduces concepts to new users of Intel's embedded solutions, and helps to increase the design efficiency of the experienced user.

1.1. Document Revision History for Embedded Design Handbook

Document Version	Changes
2020.07.22	Updated the following chapters: • Nios® II Configuration and Booting Solutions • Optimizing Nios II Based Systems and Software
2020.04.25	Updated the *Software System Design with a Nios II Processor* chapter.
2020.01.13	Updated the *Software System Design with a Nios II Processor* chapter.
2018.11.12	Updated the following chapters: • Nios II Debug, Verification, and Simulation
2017.11.06	Updated the following chapters: • Nios II Configuration and Booting Solutions • Nios II Debug, Verification, and Simulation
2017.06.12	Updated the following chapters: • Hardware System Design with Intel® Quartus® Prime and Platform Designer • Software System Design with a Nios II Processor • Nios II Debug, Verification, and Simulation • Optimizing Nios II Based Systems and Software
2016.12.19	Updated the following chapters: • First Time Designer's Guide Added new chapter: • Hardware System Design with Intel Quartus Prime and Platform Designer • Software System Design with a Nios II Processor • Nios II Configuration and Booting Solutions • Nios II Debug, Verification, and Simulation • Optimizing Nios II Based Systems and Software
2015.12.18	Updated the following chapters: • First Time Designer's Guide

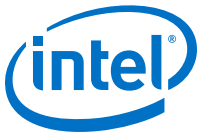
Refer to the individual chapter's revision history section for more details about the changes and previous revision history.

Related Information

• First Time Designer's Guide Revision History on page 14

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered



- Hardware System Design with Intel Quartus Prime and Platform Designer Revision History on page 118
- Software System Design with a Nios II Processor Revision History on page 217
- Nios II Configuration and Booting Solutions Revision History on page 391
- Nios II Debug, Verification, and Simulation Revision History on page 440
- Optimizing Nios II Based Systems and Software Revision History on page 499

2. First Time Designer's Guide

The First Time Designer's Guide is a basic overview of Intel 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 Intel'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
- Nios II Flash Programmer User Guide

2.1. 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 variants of the processor are available for Nios II Gen2, each with flexible features.(1)

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

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

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

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 Signal Tap 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.

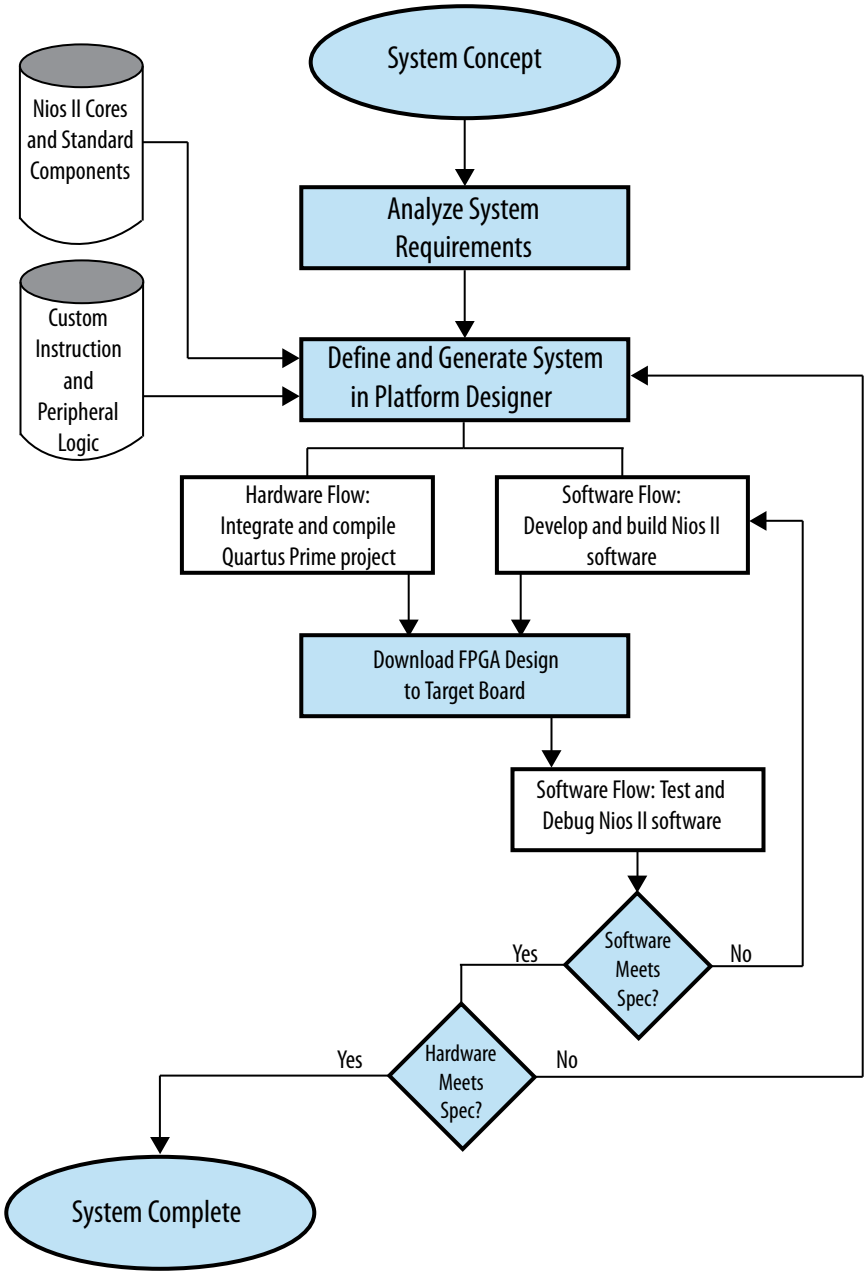
2.2. Embedded System Design

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 Intel FPGA. The "Nios II System Development Flow" section is particularly useful in helping you to decide how to approach system design using Intel's embedded hardware and software development tools. Intel 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 Platform Designer system design tool helps to manage this complexity. Even if you decide a soft-core processor does not meet your application's needs, Platform Designer 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 Intel tools to create a Nios II system.

Figure 1. General Nios II System Design Flow



Related Information

- Hardware System Design with Intel Quartus Prime and Platform Designer on page 15
- Software System Design with a Nios II Processor on page 119

2.3. Embedded Design Resources

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

2.3.1. Intel Embedded Support

Intel recommends that you seek support in the following order:

1. Look for relevant literature on the Intel Documentation page, especially on the Documentation: Nios II Processor page.
2. For reference, see the related design examples in the Intel FPGA Design Store.
3. Consult one of the following community-owned resources:
   - The Nios Forum, available on the Intel FPGA Community website.
   - The Intel FPGA Wiki website
   - Rocketboards for Linux support

   Note: Intel is not responsible for the contents of the Nios Forum and Intel FPGA Wiki websites, which are maintained by public authors and experts outside of Intel.
4. Contact technical support through the My Intel page of the Intel website to get support directly from Intel.
5. Contact your local Intel sales office or sales representative, or your field application engineer (FAE).

Related Information

- Intel FPGA Documentation
- Intel FPGA Community
- Intel FPGA Wiki
- Documentation: Nios II Processor
- Rocketboards
- Intel FPGA Design Store

2.3.2. Intel Embedded Training

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

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

Related Information

- Intel FPGA Training
- Embedded SW Designer Curriculum

2.3.3. Intel 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 Intel 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 Intel 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

2.3.4. Third Party Intellectual Property

Many third parties have participated in developing solutions for embedded designs with Intel FPGAs through the Intel DSN 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 Intel website, and select the **Ecosystem** tab; or run a search on the Intellectual Property Find IP web page..

Several community forums are also available. These forums are not controlled by Intel. The Intel FPGA 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.

For Linux support, you can also refer to Rocketboards.

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 Intel FPGA Forum or by browsing the information contained on the Intel FPGA Wiki. Many experienced developers, from Intel and elsewhere, contribute regularly to Wiki content and answer questions on the Nios Forum.

Related Information

- Intel FPGA Community
- Intel FPGA Wiki
- Intellectual Property: Find IP
- Rocketboards

2.4. Intel Embedded Glossary

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

- **Component**—A named module in Platform Designer 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 Platform Designer-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 Platform Designer 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.
- **Platform Designer**—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.

2.5. First Time Designer's Guide Revision History

Document Version	Changes
2016.12.19	• Updated the overview section: *First Time Designer's Guide*. • Moved the *Nios II Software Design* section to *Software System Design with a Nios II Processor* chapter.
2015.12.18	• Removed mention of SOPC Builder, now Platform Designer. • Removed mention of C2H Complier. • Quartus II is now Intel Quartus Prime. • Removed mention of FS2 Console. • Removed mention of Nios II IDE. • Removed section: *Nios II IDE Flow*.

Date	Version	Changes
July 2011	2.3	• Clarified this handbook does not include information about Platform Designer. • 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.
November 2008	2.0	Added System Console.
March 2008	1.0	Initial release.

Related Information

Document Revision History for Embedded Design Handbook on page 6

3. Hardware System Design with Intel Quartus Prime and Platform Designer

This chapter provides information on the hardware system design flow, designing with Platform Designer, 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 Platform Designer 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 explains how to connect signals in Platform Designer.

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

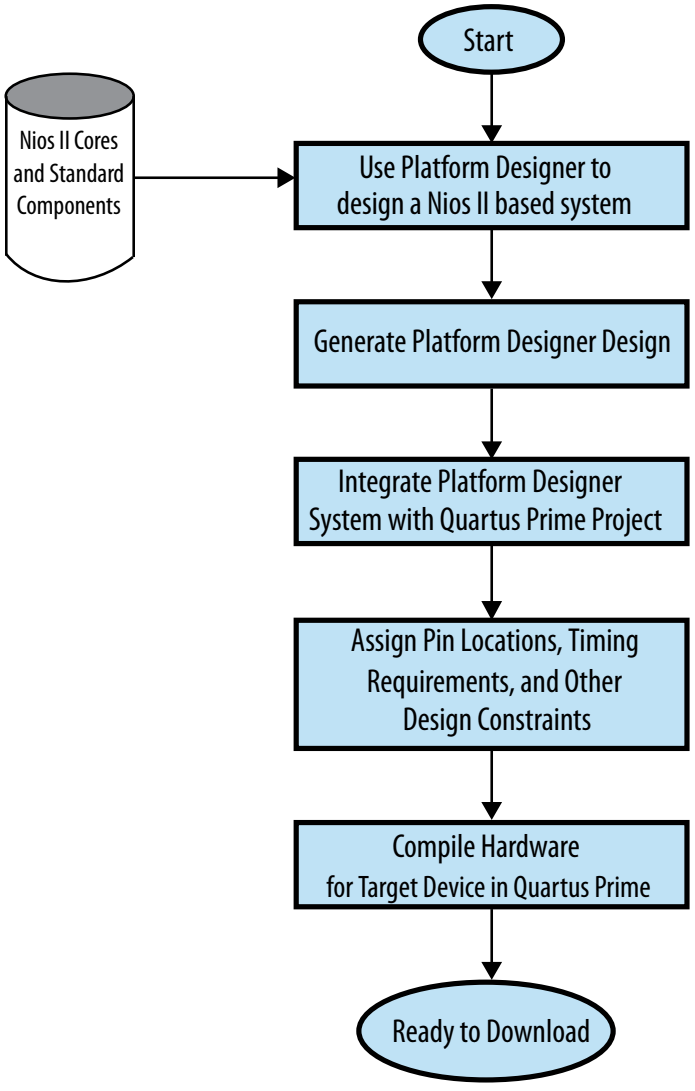
ISO 9001:2015 Registered

3.1. FPGA Hardware Design

Although you develop your FPGA-based design in Platform Designer, 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 Platform Designer system to other signals in the FPGA logic
- Constrain your design

Figure 2. Nios II System Hardware Design Flow



3.1.1. Connecting Your FPGA Design to Your Hardware

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

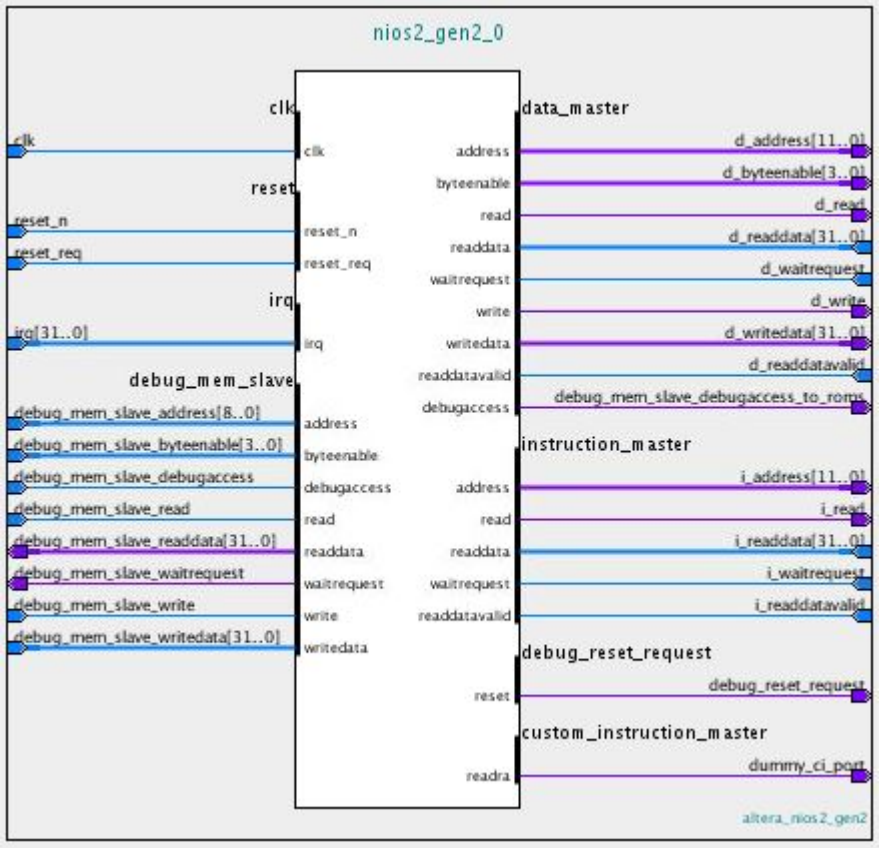
- Identify the top level of your FPGA design.
- Assign signals in the top level of your FPGA design to pins on your FPGA using any of the methods mentioned in the Intel I/O Management, Board Development Support, and Signal Integrity Analysis Resource Center page of the Intel FPGA website.

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

3.1.2. Connecting Signals to your Platform Designer System

You must define the clock and reset pins for your Platform Designer 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 Platform Designer system that includes a Nios II processor. The large symbol in this top-level diagram, labeled `std_1s40`, represents the Platform Designer system. The flag-shaped pin symbols in this diagram represent off-chip (off-FPGA) connections.

Figure 3. Top-level Block Diagram



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

3.1.3. 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 Intel Quartus Prime software uses your constraint information during design compilation to achieve Intel's best possible results.

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

3.2. System Design with Platform Designer

Platform Designer simplifies the task of building complex hardware systems on an FPGA. Platform Designer 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 Intel Quartus Prime software compiles the HDL files to create an SRAM Object File (**.sof**). For additional information about Platform Designer, refer to the Intel Quartus Prime Handbook.

Platform Designer 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 Platform Designer.

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

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

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

Platform Designer 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 Platform Designer.

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 Platform Designer, refer to *Avalon Interface Specifications*. To learn about using the component editor, refer to the Component Editor chapter in the *Intel 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, Intel 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. Intel 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 Platform Designer system. Platform Designer 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
- Design Examples in the Intel Design Store
- Nios II Gen2 Hardware Development Tutorial
- Avalon Interface Specifications
- Verification and Board Bring-Up on page 418

3.2.1. Intel System on a Programmable Chip (Platform Designer) Solutions

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

A Platform Designer 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 Platform Designer system.

In Intel Platform Designer solutions, the hardware design is implemented in an Intel FPGA device. An Intel FPGA device is volatile—contents are lost when the power is turned off— and reprogrammable. When an Intel FPGA is programmed, the logic cells inside it are configured and connected to create a Platform Designer 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 Platform Designer—basic properties:
  - **Volatility**—The FPGA is functional only after it is configured, and it can be reconfigured at any time.
  - **Design**—Many Intel Platform Designer systems are designed using Platform Designer and the Intel Quartus Prime software, and may include multiple peripherals and processors.
  - **Configuration**—FPGA configuration can be performed through a programming cable, such as the Intel FPGA Download 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 Platform Designer, 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 Intel FPGA Download 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.

3.2.2. Platform Designer Design

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

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

  The **.qsf** file contains all of the device, pin, timing, and compilation settings for the Intel 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**)

    Platform Designer 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 Intel Quartus Prime Help.
- Platform Designer Design File (**.qsys**)
- Platform Designer Information File (**.sopcinfo**)

  This file contains an XML description of your Platform Designer system. Platform Designer 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 Intel Quartus Prime software, Platform Designer, 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 *Intel Quartus Prime Handbook Volume 1: Design and Synthesis*.

3.3. Interfacing an External Processor to an Intel FPGA

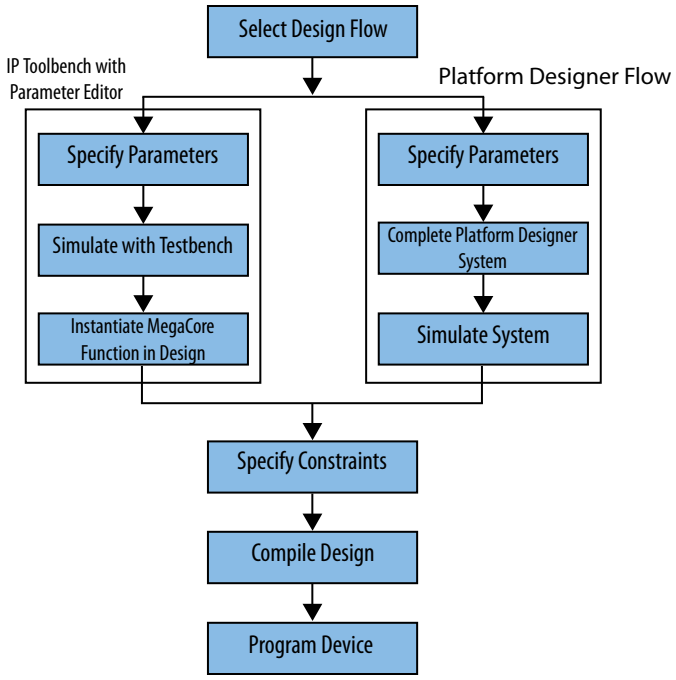
Intel provides options to connect an external processor to an Intel FPGA. These interface options include the PCI Express*, PCI, RapidIO™, serial peripheral interface (SPI) interface or a simple custom bridge that you can design yourself.

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 Intel'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 Intel FPGA IP functions using either the Parameter Editor or Platform Designer design flow. The PCI Lite and SPI cores are only available in the Platform Designer design flow. Platform Designer 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 Platform Designer. The figure below shows the steps you take to instantiate a component in both design flows.

Figure 4. Platform Designer and Parameter Editor Design Flows



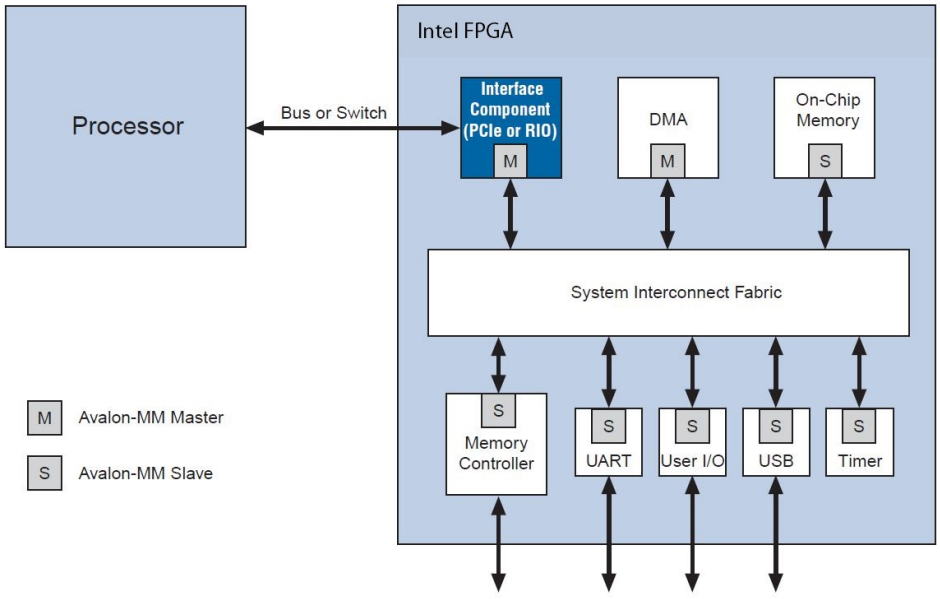
The remainder of this section provides an overview of the Intel FPGA IP functions that you can use to interface an Intel FPGA to an external processor. It covers the following topics:

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

3.3.1. Configuration Options

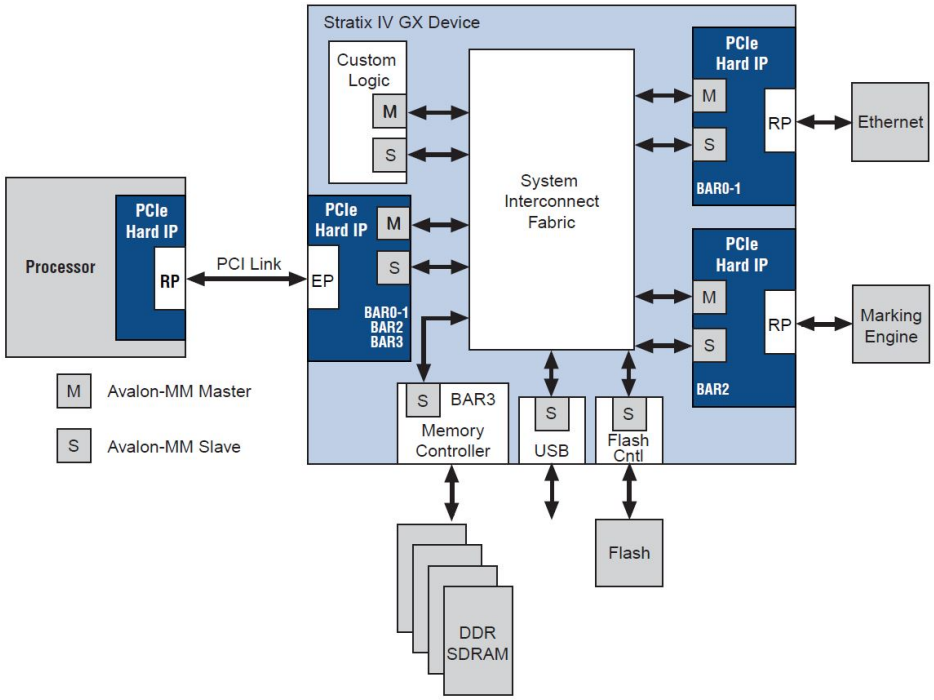
The figure below illustrates a Platform Designer system design that includes a high-performance external bus or switch to connect an industry-standard processor to an external interface of an IP core inside the FPGA. This IP core also includes an Avalon-MM master port that connects to the Platform Designer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

Figure 5. FPGA with a Bus or Switch Interface Bridge for Peripheral Expansion



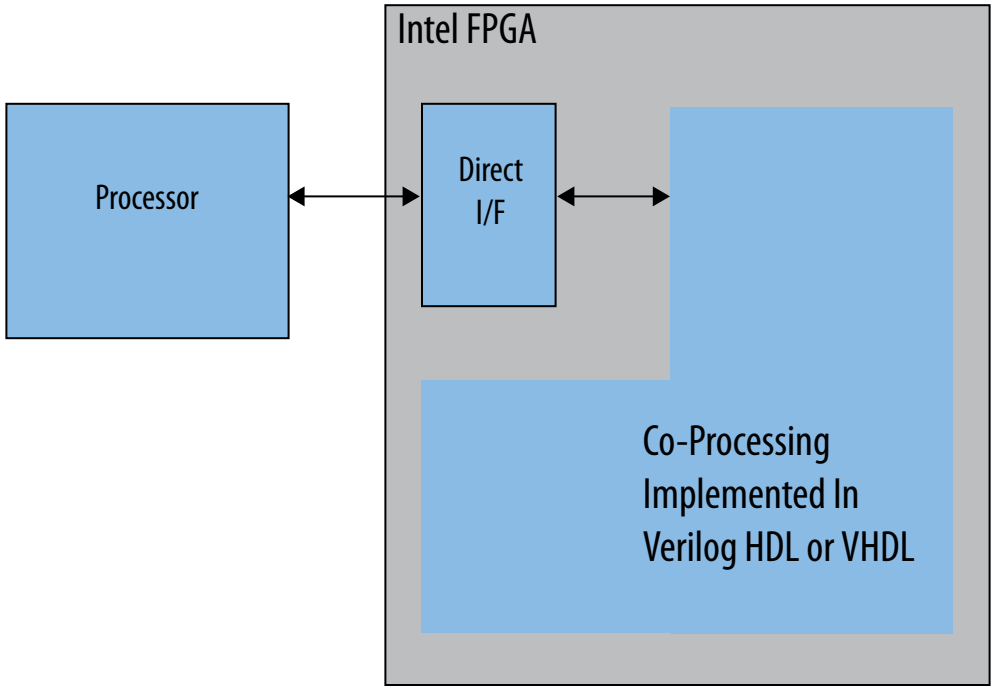
The design below includes an external processor that interfaces to a PCI Express endpoint inside the FPGA. The system interconnect fabric inside the implements a partial crossbar switch between the endpoint that connects to the external processor and two additional PCI Express root ports that interface 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. Platform Designer automatically generates the system interconnect fabric to connect the components in the system.

Figure 6. FPGA with a Processor Bus or SPI for Peripheral Expansion



Alternatively, you can also implement your logic in Verilog HDL or VHDL without using Platform Designer. 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 7. FPGA Performs Co-Processing



The table below summarizes the components Intel provides to connect an Intel 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 Platform Designer. Alternative implementations of these components are also available through the Intel IP Core Partners Program (DSN) partners. The partners offer a broad portfolio of IP cores optimized for Intel devices.

For a complete list of third-party IP for Intel FPGAs, refer to the Intellectual Property: Find IP web page of the Intel website.

Table 1. Processor Interface Solutions Available from an Intel Device

Protocol	Available in Platform Designer	Available In Parameter Editor	Third-Party Solution	Intel FPGA IP Evaluation Mode Available
RapidIO	Yes	Yes	Yes	Yes
PCI Express	Yes	Yes	Yes	Yes
PCI Lite	Yes	—	—	License not required
SPI	Yes	—	—	

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

Table 2. Partial list of some of the peripheral interfaces available for Platform Designer

Protocol	Available in Platform Designer	Available In Parameter Editor	Third-Party Solution	Intel FPGA IP Evaluation Mode 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 Platform Designer refer to the *Embedded Peripherals IP User Guide* and the Intellectual Property: Find IP page.

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
- Using Intel FPGA IP Evaluation Mode

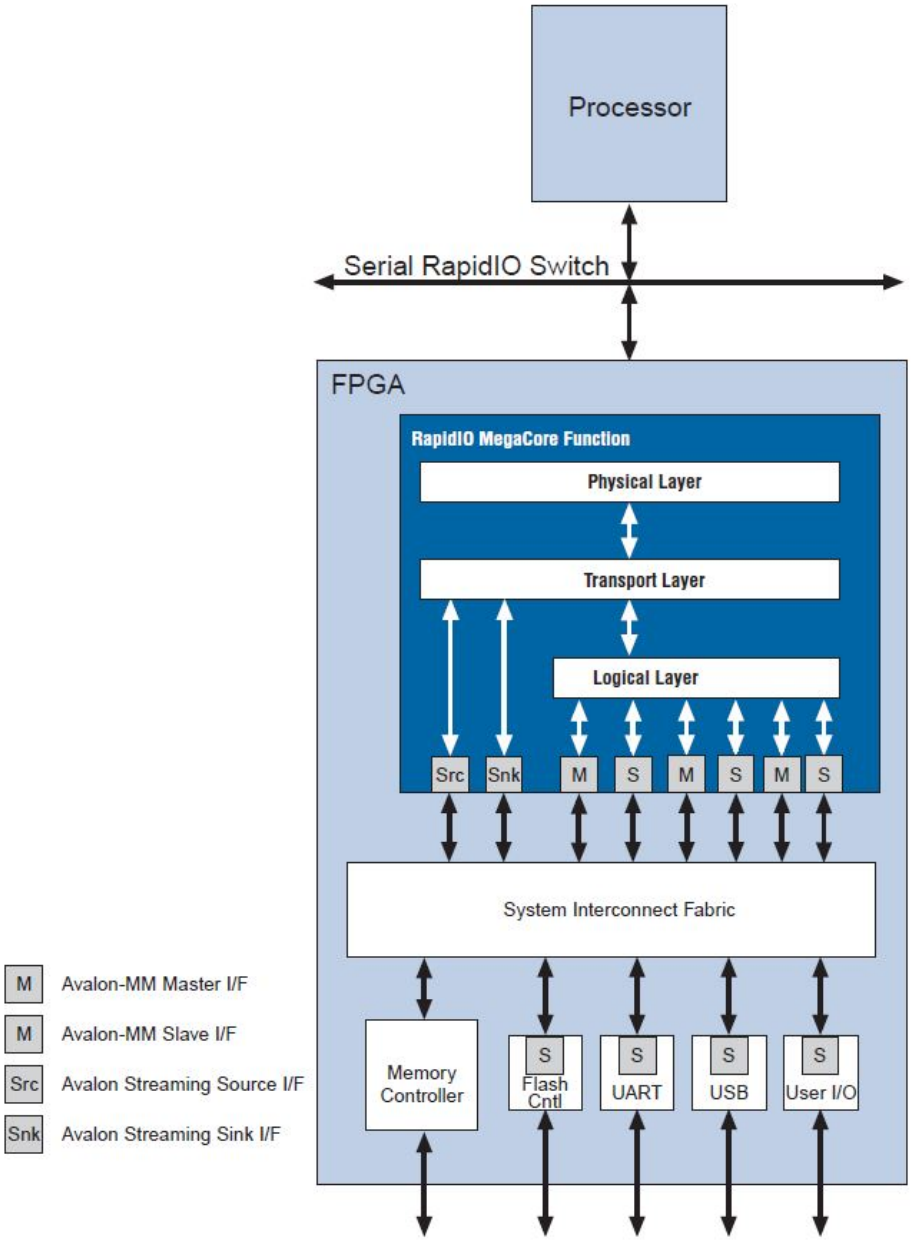
3.3.2. RapidIO Interface

RapidIO is a high-performance packet-switched protocol that transports data and control information between processors, memories, and peripheral devices. The RapidIO Intel FPGA IP function is available in Platform Designer includes Avalon-MM ports that translate Serial RapidIO transactions into Avalon-MM transactions. The Intel FPGA IP 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 Platform Designer design flow, you can integrate a RapidIO endpoint in a Platform Designer system. You connect the ports using the Platform Designer **System Contents** tab and Platform Designer automatically generates the system interconnect fabric. The figure below illustrates a Platform Designer system that includes a processor and a RapidIO Intel FPGA IP function.

Figure 8. 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 Intel FPGA IP function: *RapidIO Intel FPGA IP Function User Guide* and *AN513: RapidIO Interoperability With TI 6482 DSP Reference Design*.

Related Information

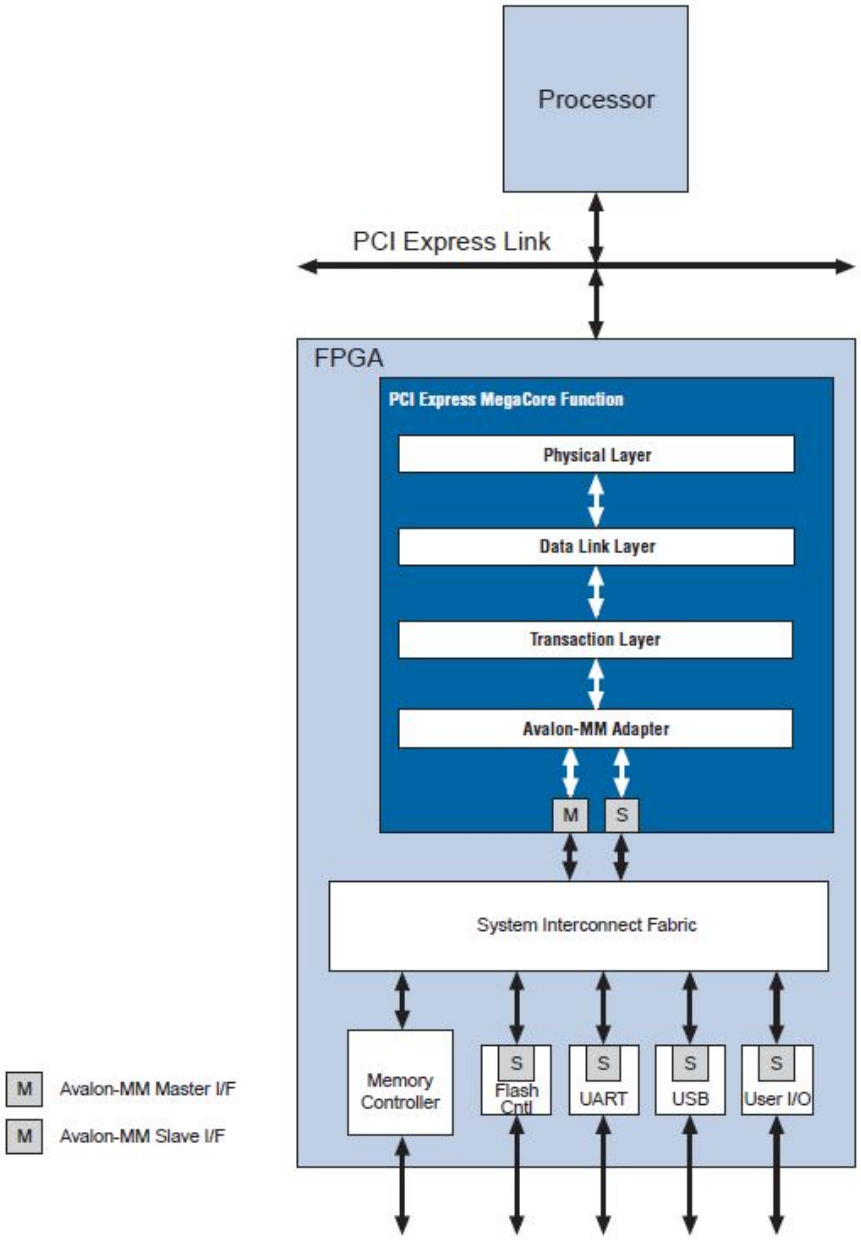
- RapidIO.org
- RapidIO IP Core User Guide
- RapidIO Interoperability with TI 6482 DSP Reference Design

3.3.3. PCI Express Interface

The Intel IP Compiler for PCI Express configured using the Platform Designer 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 Platform Designer components. The figure below illustrates a design that links an external processor to an Platform Designer 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 Intel FPGA devices as a hard IP implementation and can be used as a root port or end point. I

Figure 9. Example System with PCI Express Interface



The figure shows an example system in which an external processor communicates with an Intel 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 PCIe User Guide
- AN456: PCI Express High Performance Reference Design

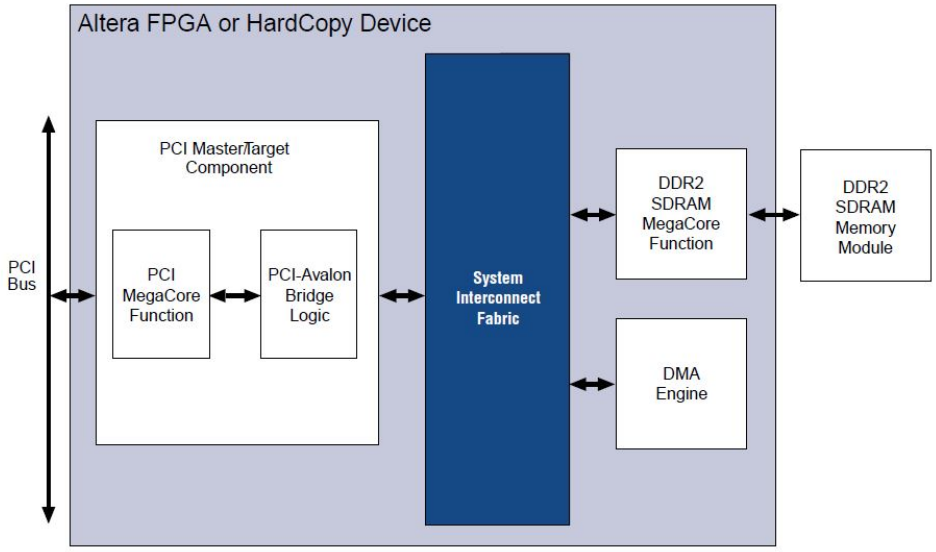
- External PHY Support in PCI Express MegaCore Functions
- PCI Express to External Memory Reference Design

3.3.4. PCI Interface

Intel 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 Intel FPGA IP function using the Parameter Editor or Platform Designer design flow.

The PCI Platform Designer 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 Platform Designer system using the PCI Intel FPGA IP function. You can parameterize the PCI Intel FPGA IP function with a 32- or 64-bit interface.

Figure 10. PCI Intel FPGA IP Function in a Platform Designer System



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

Related Information

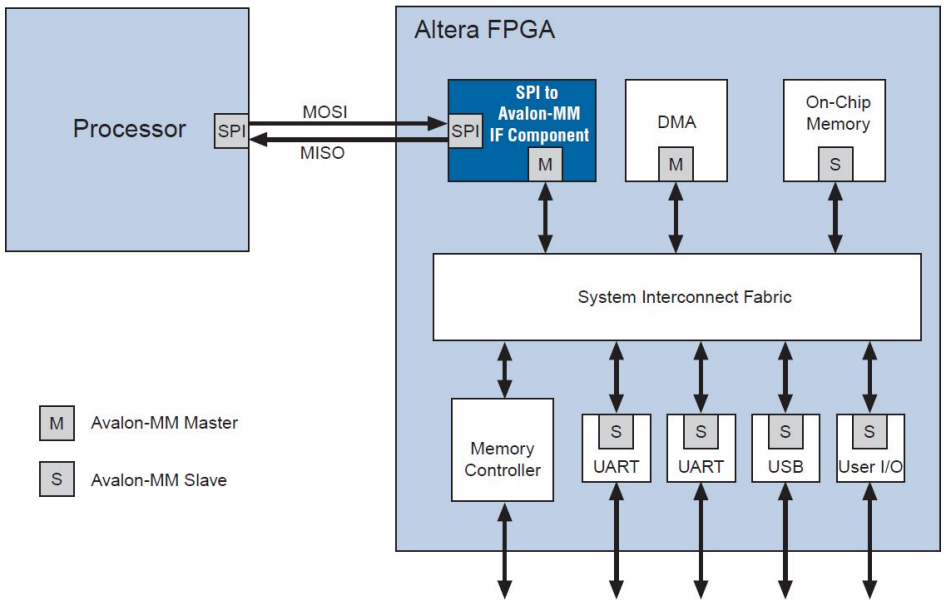
PCI Compiler User Guide

3.3.5. Serial Protocol Interface (SPI)

The SPI Slave to Avalon Master Bridge component provides a simple connection between processors and Platform Designer 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 Platform Designer system for memory access and peripheral expansion.

The SPI Slave to Avalon Master Bridge is an Platform Designer-ready component that integrates easily into any Platform Designer 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 11. 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 Platform Designer 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 Platform Designer 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

3.3.6. 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 Platform Designer 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 Platform Designer 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 Platform Designer Components chapter of the *Intel 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.

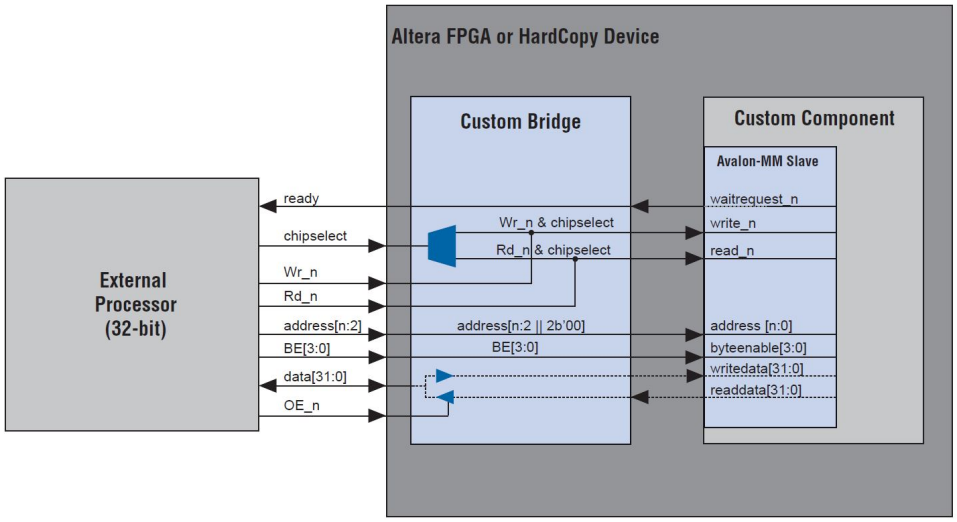
Consider the following points when designing a custom bridge to interface between an external processor and the Avalon-MM interface:

- 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 Platform Designer 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 12. 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

3.4. 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.

Intel recommends understanding the following documents before proceeding:

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

Related Information

Avalon Interface Specifications

3.4.1. 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 43
- ARM BE-32 Bus Byte Ordering on page 45

3.4.1.1. 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.

3.4.1.2. 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 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 47
- System-Wide Arithmetic Byte Reordering in Software on page 50

3.4.2. 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 Platform Designer Interconnect chapter of the *Intel Quartus Prime Handbook Volume 1: Design and Synthesis*

Related Information

Avalon Interface Specifications

3.4.2.1. Dynamic Bus Sizing DMA Examples

A direct memory access (DMA) engine moves memory contents from a source location to a destination location. Because Platform Designer 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 13. 16-Bit to 32-Bit Memory DMA Transfer

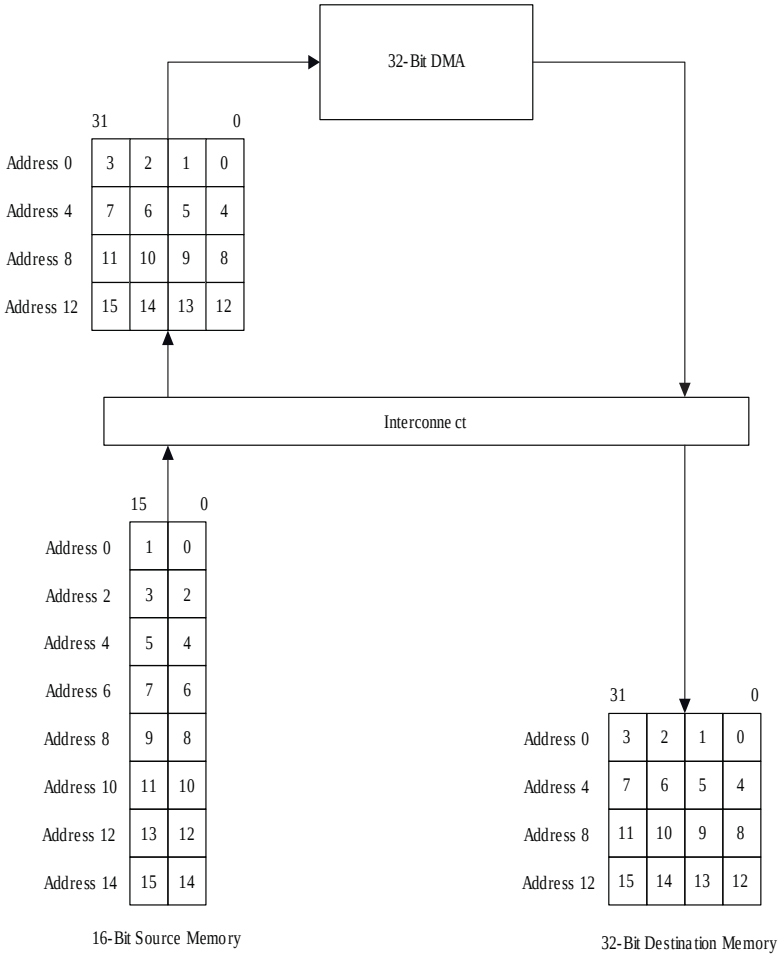


Figure 14. 32-Bit to 64-Bit Memory DMA Transfer

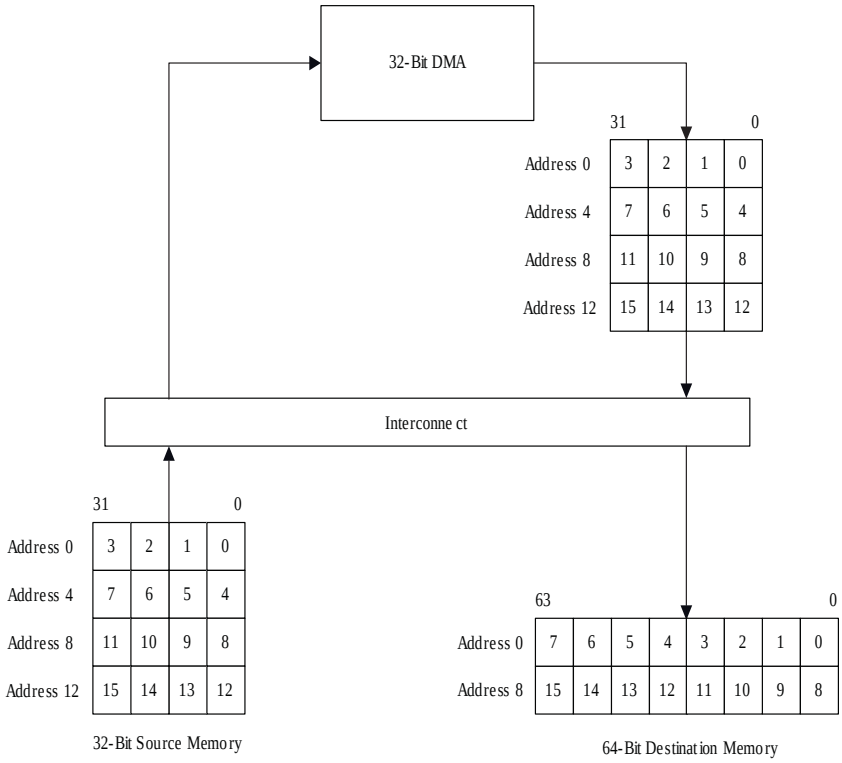
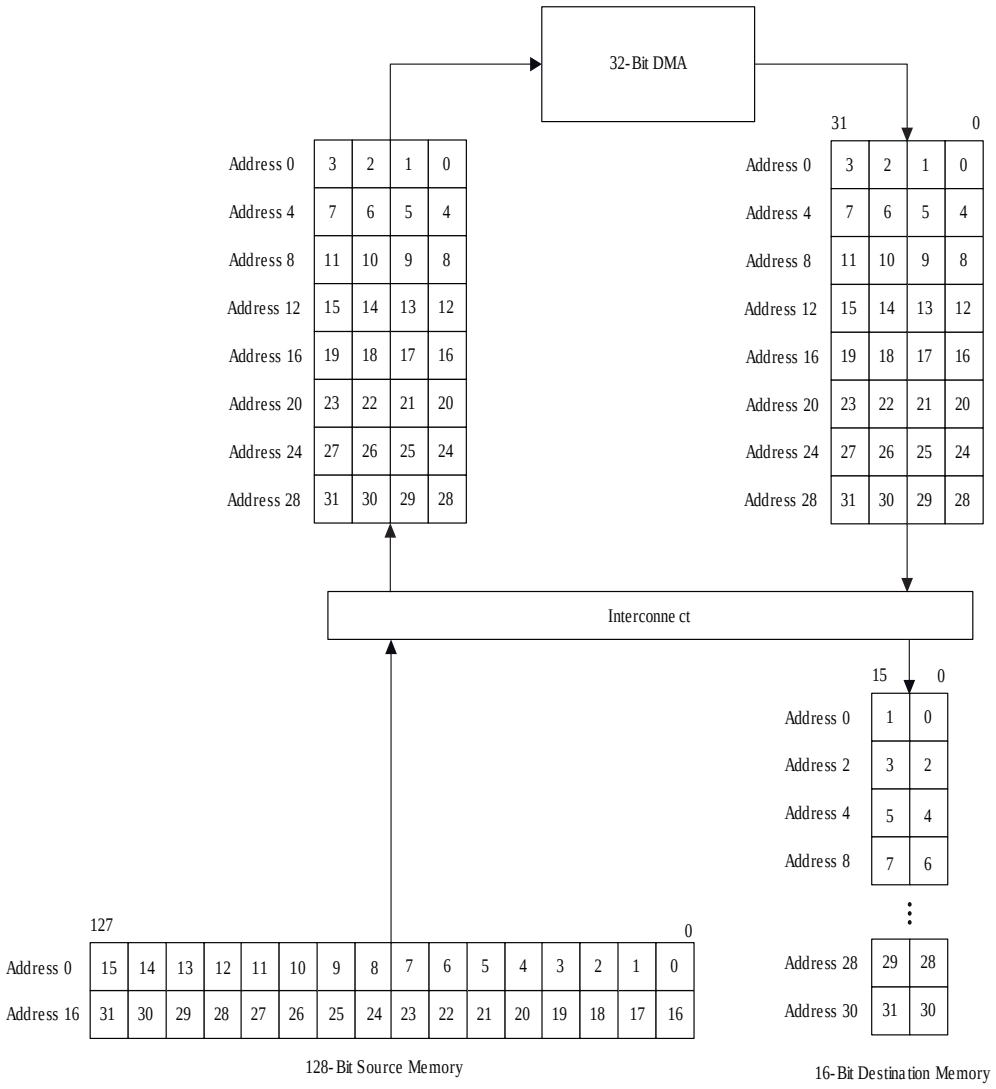


Figure 15. 128-Bit to 16-Bit Memory DMA Transfer

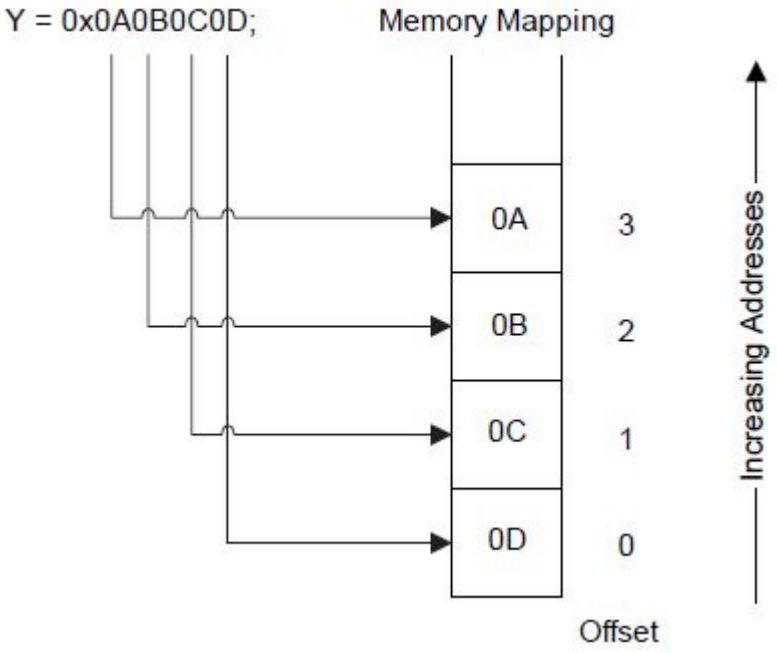


3.4.3. Nios II Processor Data Accesses

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 16. 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. 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	—	—
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 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);
```

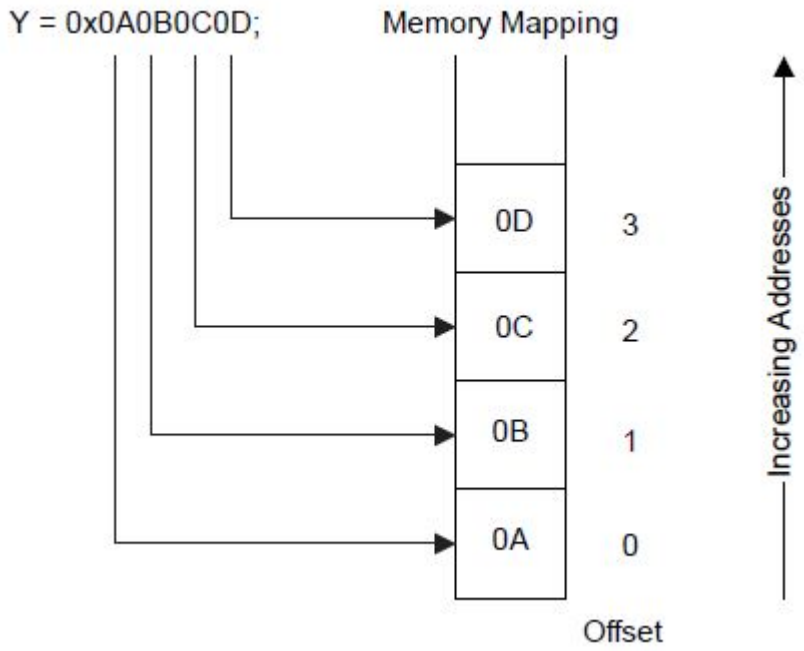
3.4.4. Adapting Processor Masters to be Avalon-MM Compliant

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. Intel 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 17. 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"

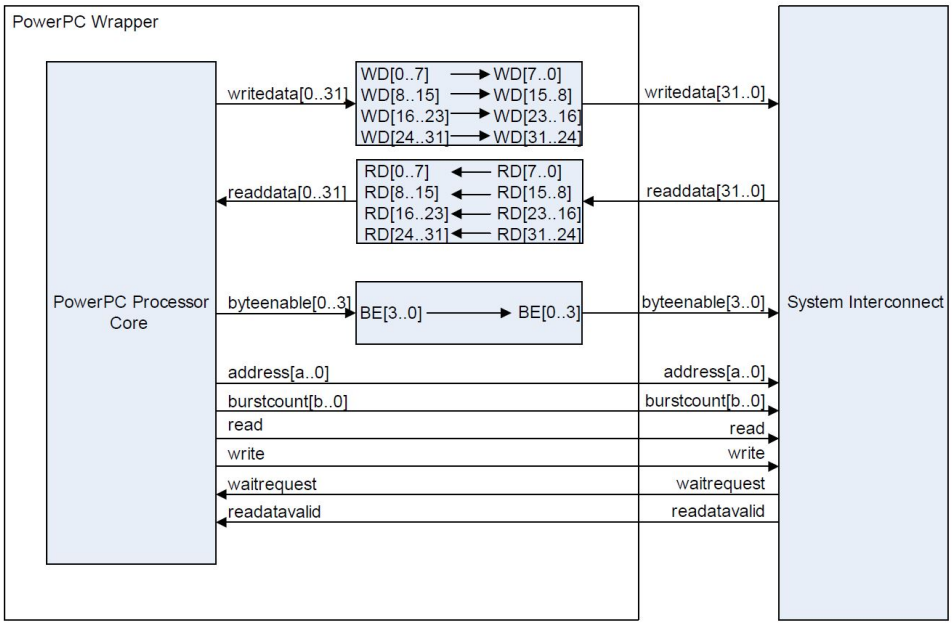
Related Information

- Nios II Processor Data Accesses on page 40
- PowerPC Bus Byte Ordering on page 43
- ARM BE-32 Bus Byte Ordering on page 45

3.4.4.1. 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 18. 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.

3.4.4.2. 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.

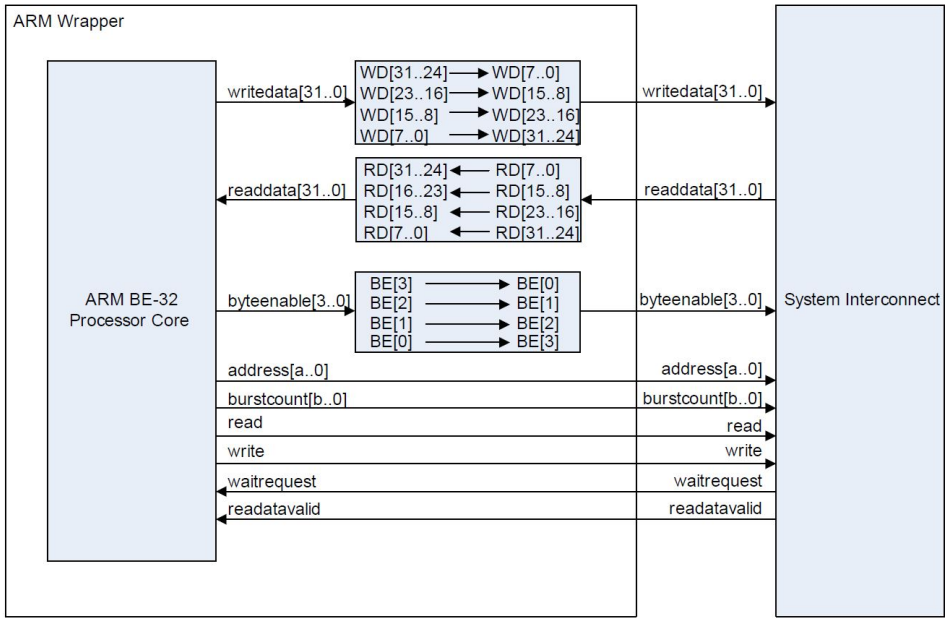
Table 4. ARM BE-32 Write Data Mapping

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 on page 41. 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 19. 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.

3.4.4.3. 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 47

3.4.4.4. 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.

3.4.4.5. 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 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

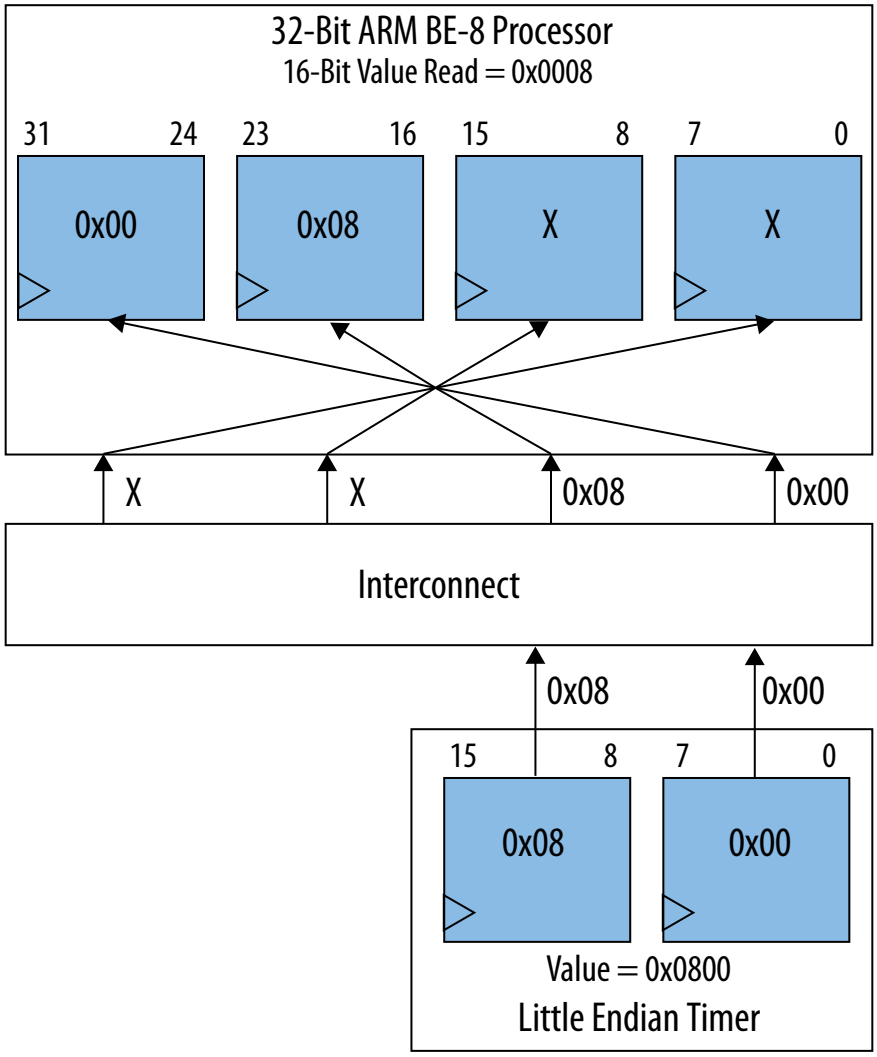
The big endian ARM BE-8 mapping in the table above matches the little endian Nios II processor mapping for all single byte accesses. If you ensure that your processor is Avalon-MM compliant, you can easily share individual bytes of data between big and little endian processors and peripherals.

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 20. ARM BE-8 Processor Accessing a Little Endian Peripheral



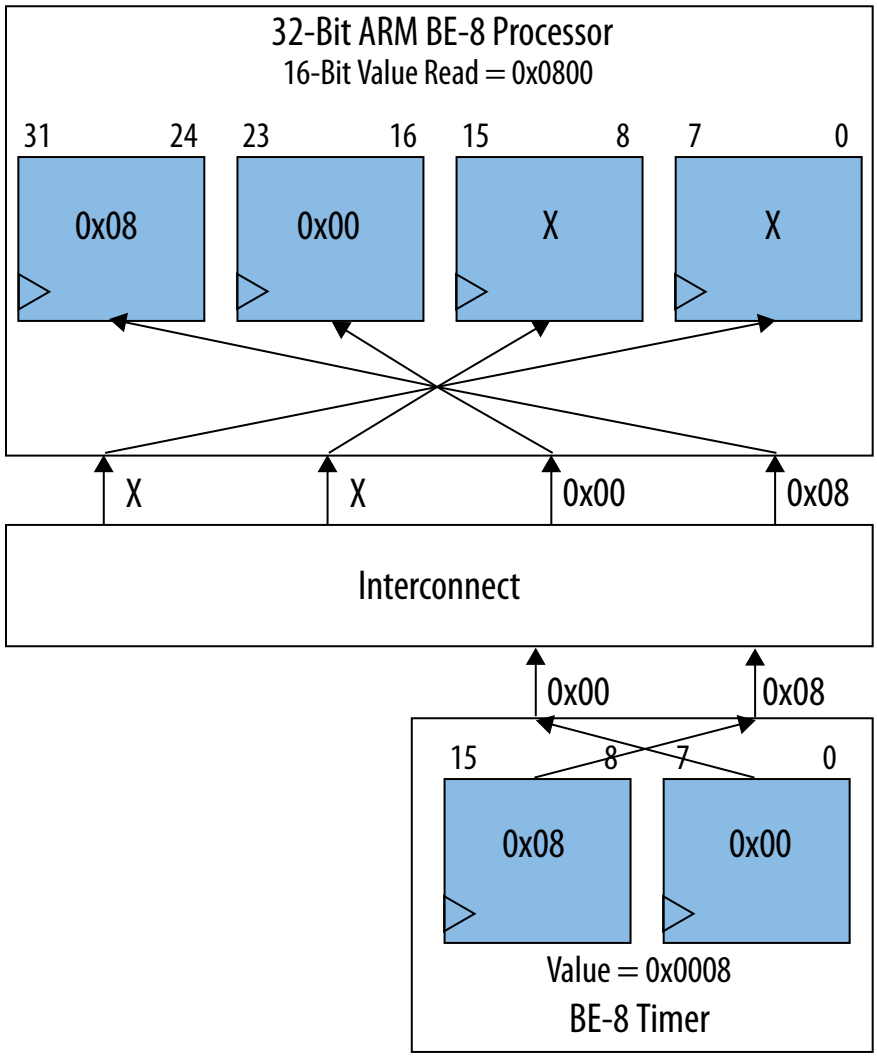
For the values to be interpreted accurately, the processor must either read each byte lane individually and then combine the two byte reads into a single 16-bit value in software, or read the single 16-bit value and swap the bytes in software.

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 21. ARM BE-8 Processor Accessing a BE-8 Peripheral



3.4.5. 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.

3.4.5.1. 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 Intel that contains an Avalon-MM master or slave port uses little endian arithmetic byte ordering. If your system consists primarily of components provided by Intel, 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.

Intel 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.

3.4.5.2. 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. 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.

3.5. Memory System Design

This section describes the efficient use of memories in a Platform Designer 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.

3.5.1. 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.

3.5.1.1. 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.

3.5.1.2. 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.

3.5.2. 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 Platform Designer. 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.

3.5.2.1. 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.

3.5.2.2. 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 Intel Quartus Prime Help.

3.5.2.3. Best Applications

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

3.5.2.3.1. 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.

3.5.2.3.2. 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*.

3.5.2.3.3. 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.

3.5.2.3.4. 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*.

3.5.2.4. 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.

3.5.2.5. 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.

3.5.2.6. 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.

3.5.3. 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.

3.5.3.1. 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.

3.5.3.2. 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.

3.5.3.3. 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.

3.5.3.4. 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.

3.5.3.5. 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.

3.5.3.6. 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 accesses 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.

3.5.4. 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.

3.5.4.1. 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.

3.5.4.2. 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.

3.5.4.3. 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)

3.5.4.4. 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 writable 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.

3.5.4.5. 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. Intel 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 Intel 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 Intel FPGAs; however, Intel does not provide any built-in support.

3.5.5. 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 Intel FPGAs, refer to the *External Memory Interface Handbook*.

3.5.5.1. 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.

3.5.5.2. 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.

3.5.5.3. 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.

3.5.5.4. 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.

3.5.5.5. 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*.

3.5.5.6. SDRAM Controller Types Available From Intel

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

Table 6. Memory Controller Available from Intel

Controller Name	Description
SDR SDRAM Controller	This controller is the only SDR SDRAM controller Intel 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. Intel does not recommend it for new designs.
High Performance DDR/DDR2 Controller	This controller is the DDR/DDR2 controller that Intel recommends for new designs. It supports two primary clocking modes, full-rate and half-rate. • Full-rate mode presents data to the Platform Designer system at twice the width of the actual DDR SDRAM device at the full SDRAM clock rate. • Half-rate mode presents data to the Platform Designer 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 Intel 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*.

3.5.5.7. 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.

3.5.5.7.1. 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 Platform Designer 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.

3.5.5.7.2. 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.

3.5.5.7.3. 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.

3.5.5.7.4. 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.

3.5.5.7.5. 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.

3.5.5.7.6. 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.

3.5.6. Case Study

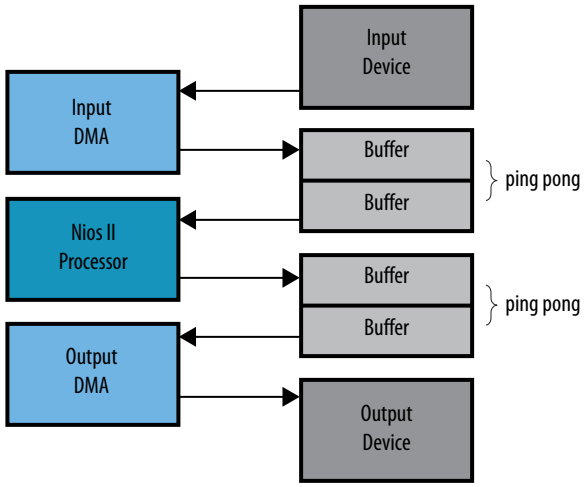
This section describes the optimization of memory partitioning in a video processing application to illustrate the concepts discussed earlier.

3.5.6.1. 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 22. Sample Application Architecture

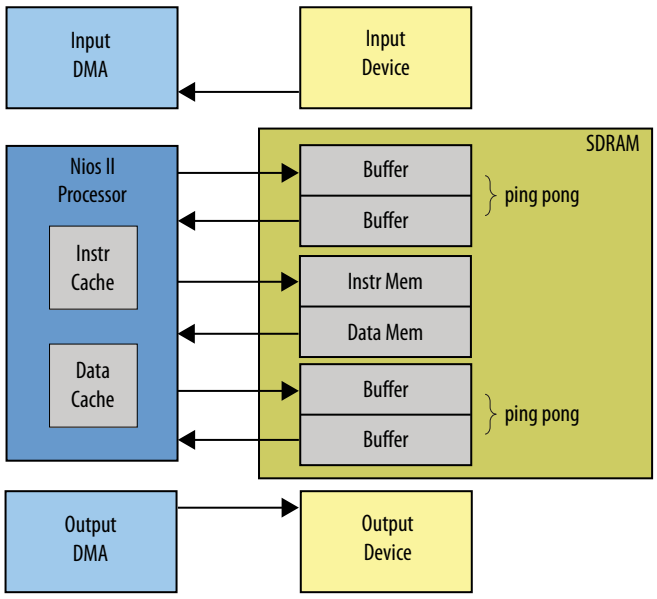


3.5.6.2. Initial Memory Partitioning

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 23. 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 these 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.

3.5.6.3. Optimized Memory Partitioning

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

3.5.6.3.1. Add an External SRAM for Input Buffers

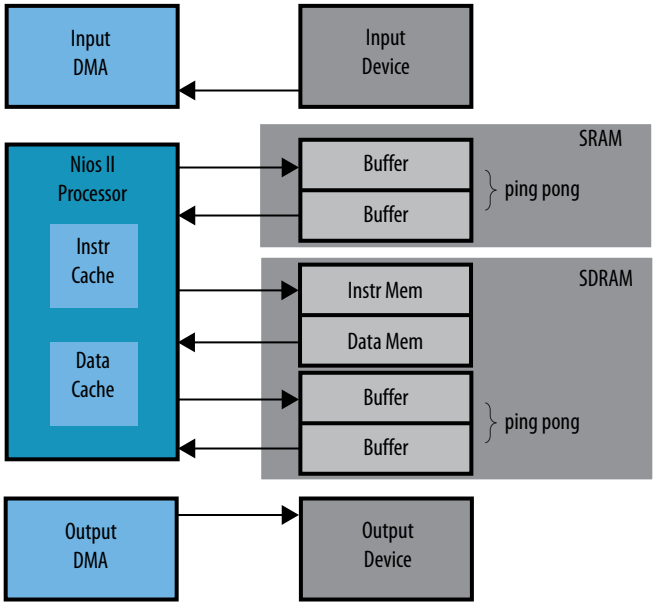
The first optimization to improve efficiency is to move the input buffering from the SDRAM to an external SRAM device. This technique creates performance gains for the following three reasons:

- 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 24. 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

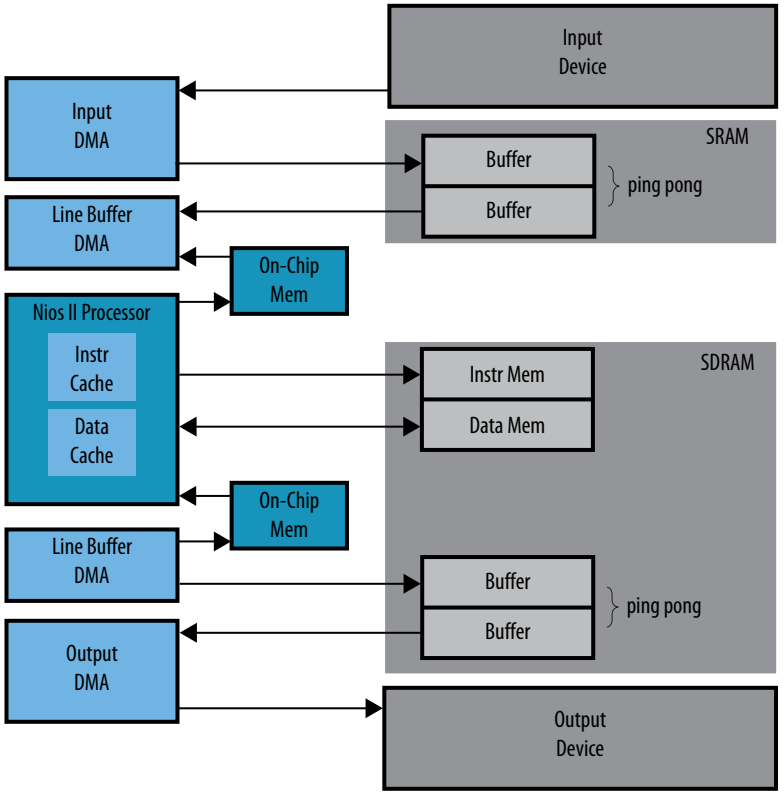
caches, the SDRAM bandwidth required by the processor is likely to be relatively small. Therefore, sharing the SDRAM for processor instructions, processor data, and the video output channel is probably acceptable. If necessary, increasing the processor cache sizes can further reduce the processor's reliance on SDRAM bandwidth.

3.5.6.3.2. 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 25. On-Chip Memories Added As Line Buffers



3.6. Nios II Hardware Development Tutorial

This tutorial describes the system development flow for the Intel Nios II processor.

Using the Intel Quartus Prime software and the Nios II Embedded Design Suite (EDS), you can:

- build a Nios II hardware system design
- create a software program that runs on the Nios II system and interfaces with components on Intel development boards

Building embedded systems in FPGAs involves system requirements analysis, hardware design tasks, and software design tasks. This tutorial guides you through the basics of each topic, with special focus on the hardware design steps.

3.6.1. Software and Hardware Requirements

The following are the software requirements for the tutorial:

- Intel Quartus Prime software version 14.0 or later—The software must be installed on a Windows or Linux computer that meets the Intel Quartus Prime minimum requirements.
- Nios II EDS version 14.0 or later.
- Design files for the design example—Refer related information below for the design example file.

You can build the design example with any Intel development board or your own custom board that meets the following hardware requirements:

- The board must have either Intel MAX® 10, Stratix series, Cyclone series, or Arria series FPGA.
- The FPGA must contain a minimum of 2800 logic elements (LE) or adaptive lookup tables (ALUT).
- The FPGA must contain a minimum of 40 M9K memory blocks.
- An oscillator must drive a constant clock frequency to an FPGA pin. The maximum frequency limit depends on the speed grade of the FPGA. Frequencies of 50 MHz or less should work for most boards; higher frequencies might work.
- FPGA I/O pins can optionally connect to eight or fewer LEDs to provide a visual indicator of processor activity.
- The board must have a JTAG connection to the FPGA that provides a programming interface and communication link to the Nios II system. The JTAG connection can be a dedicated 10-pin JTAG header for an Intel FPGA USB Download Cable or a USB connection with Intel FPGA Download Cable circuitry embedded on the board.

Note: Refer to the documentation for your board that describes clock frequencies and pinouts. For Intel development boards, refer to the related information below.

Related Information

- The Intel FPGA Software Installation and Licensing manual
- All Development Kits
- Nios II Hardware Development Design Example Webpage

3.6.2. Intel FPGA IP Evaluation Mode

You can perform this tutorial on hardware without a license. With Intel FPGA IP Evaluation Mode, you can perform the following actions:

- Simulate the behavior of a Nios II processor within your system
- Verify the functionality of your design
- Evaluate the size and speed of your design quickly and easily
- Generate time-limited device programming files for designs that include Nios II processors
- Program a device and verify your design in hardware

You need to purchase a license for the Nios II processor only when you are completely satisfied with its functionality and performance, and want to use your design in production.

Related Information

Using Intel FPGA IP Evaluation Mode

3.6.3. Nios II Design Example

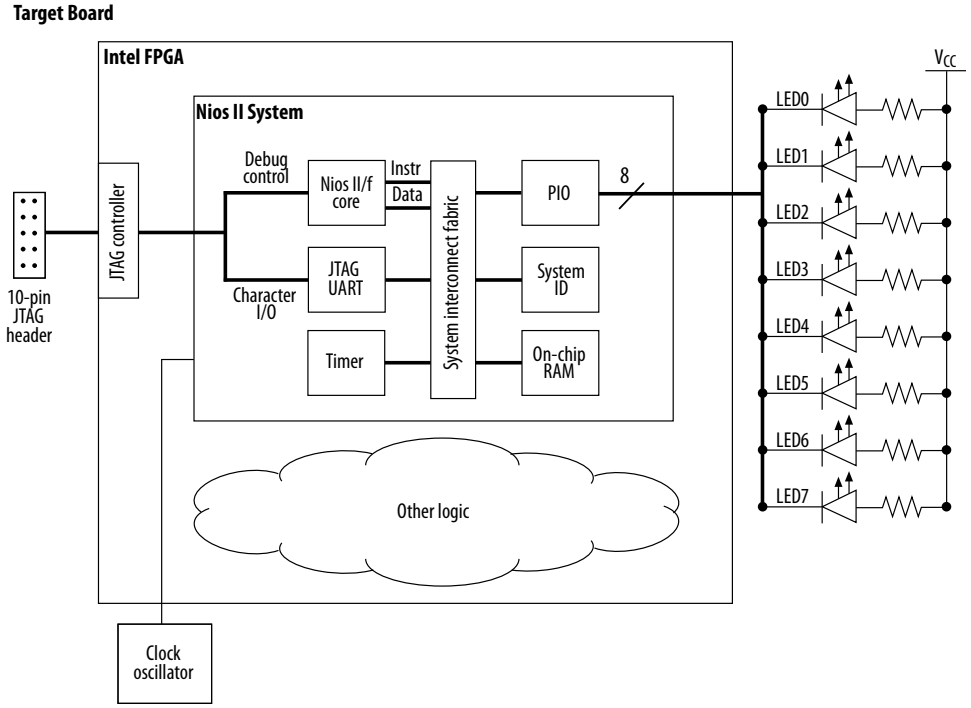
The design example you build in this tutorial demonstrates a small Nios II system for control applications, that displays character I/O output and blinks LEDs in a binary counting pattern. This Nios II system can also communicate with a host computer, allowing the host computer to control logic inside the FPGA.

The example Nios II system contains the following components:

- Nios II/f processor core
- On-chip memory
- Timer
- JTAG UART
- 8-bit parallel I/O (PIO) pins to control the LEDs
- System identification component

Figure 26. Nios II Design Example Block Diagram

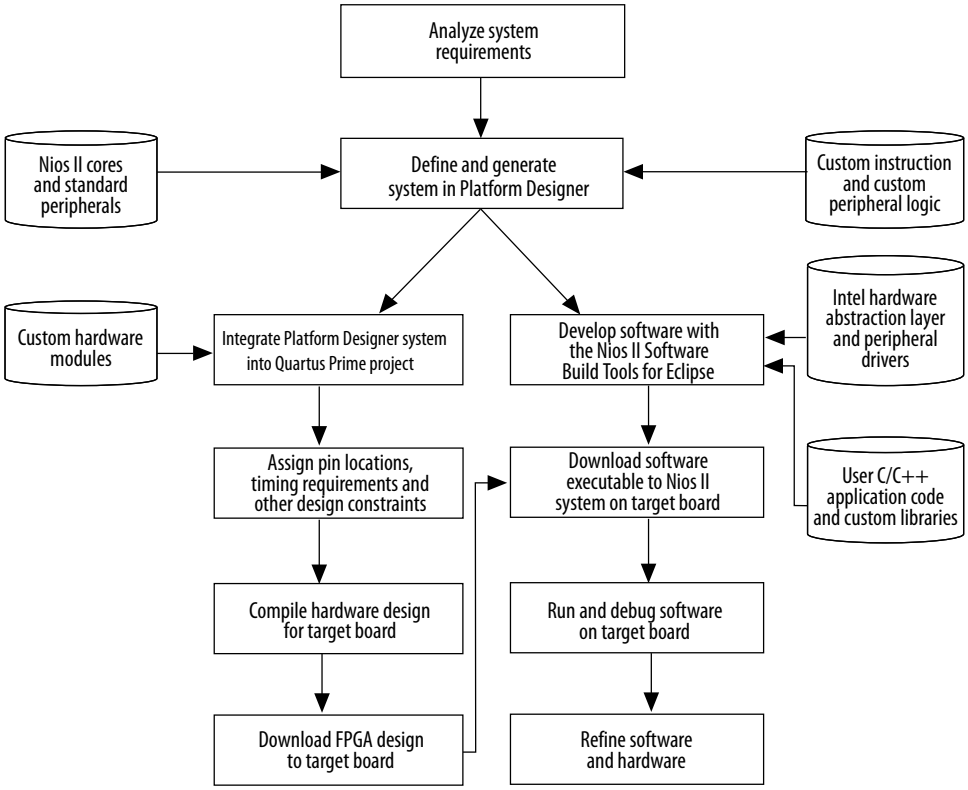
The block diagram shows the relationship between the host computer, the target board, the FPGA, and the Nios II system.



Other logic can exist within the FPGA alongside the Nios II system. In fact, most FPGA designs with a Nios II system also include other logic. A Nios II system can interact with other on-chip logic, depending on the needs of the overall system. This design example does not include other logic in the FPGA.

3.6.4. Nios II System Development Flow

Figure 27. Nios II System Development Flow



The Nios II development flow consists of three types of development:

- hardware design steps
- software design steps
- system design steps, involving both hardware and software

The design steps in this tutorial focus on hardware development, and provide only a simple introduction to software development.

3.6.4.1. Analyzing System Requirements

The development flow begins with predesign activity which includes an analysis of the application requirements, such as the following questions:

- What computational performance does the application require?
- How much bandwidth or throughput does the application require?
- What types of interfaces does the application require?
- Does the application require multithreaded software?

Based on the answers to these questions, you can determine the concrete system requirements, such as:

- Which Nios II processor core to use: smaller or faster.
- What components the design requires and how many of each kind.
- Which real-time operating system (RTOS) to use, if any.
- Where hardware acceleration logic can dramatically improve system performance. For example:
  - Could adding a DMA component eliminate wasted processor cycles copying data?
  - Could a custom instruction replace the critical loop of a DSP algorithm?

Analyzing these topics involve both the hardware and software point of view.

3.6.4.2. Defining and Generating the System in Platform Designer

After analyzing the system hardware requirements, you use Platform Designer to specify the Nios II processor core(s), memory, and other components your system requires. Platform Designer automatically generates the interconnect logic to integrate the components in the hardware system.

You can select from a list of standard processor cores and components provided with the Nios II EDS. You can also add your own custom hardware to accelerate system performance. You can add custom instruction logic to the Nios II core which accelerates CPU performance, or you can add a custom component which offloads tasks from the CPU. This tutorial covers adding standard processor and component cores, and does not cover adding custom logic to the system.

The primary outputs of Platform Designer are the following file types:

Table 7. Platform Designer Primary Output File Types

File Types	Description
Platform Designer Design File (`.qsys`)	Contains the hardware contents of the Platform Designer system
SOPC Information File (`.sopcinfo`)	Contains a description of the contents of the .qsys file in Extensible Markup Language File (`.xml`) format. The Nios II EDS uses the .sopcinfo file to create software for the target hardware.
Hardware description language (`HDL`) files	Are the hardware design files that describe the Platform Designer system. The Intel Quartus Prime software uses the HDL files to compile the overall FPGA design into an SRAM Object File (`.sof`).

Related Information

Intel Quartus Prime Handbook

3.6.4.3. Integrating the Platform Designer System into the Intel Quartus Prime Project

After generating the Nios II system using Platform Designer, you integrate it into the Intel Quartus Prime project. Using the Intel Quartus Prime software, you perform all tasks required to create the final FPGA hardware design.

Using the Intel Quartus Prime software, you can:

- assign pin locations for I/O signals
- specify timing requirements
- apply other design constraints
- compile the Intel Quartus Prime project to produce a `.sof` to configure the FPGA

You download the `.sof` to the FPGA on the target board using an Intel download cable, such as the Intel FPGA USB Download Cable. After configuration, the FPGA behaves as specified by the hardware design, which in this case is a Nios II processor system.

3.6.4.4. Developing Software with the Nios II Software Build Tools for Eclipse

You can perform all software development tasks for your Nios II processor system using the Nios II Software Build Tools (SBT) for Eclipse™.

After you generate the system with Platform Designer, you can begin designing your C/C++ application code immediately with the Nios II SBT for Eclipse. Intel provides component drivers and a hardware abstraction layer (HAL) which allows you to write Nios II programs quickly and independently of the low-level hardware details. In addition to your application code, you can design and reuse custom libraries in your Nios II SBT for Eclipse projects.

To create a new Nios II C/C++ application project, the Nios II SBT for Eclipse uses information from the `.sopcinfo` file. You also need the `.sof` file to configure the FPGA before running and debugging the application project on target hardware.

The Nios II SBT for Eclipse can produce several outputs, listed below. Not all projects require all of these outputs.

Table 8. Nios II SBT for Eclipse Outputs

The Nios II SBT for Eclipse can produce several outputs but not all projects require all of these outputs.

Output	Description
`system.h` file	• Defines symbols for referencing the hardware in the system. • The Nios II SBT for Eclipse automatically create this file when you create a new board support package (BSP).
Executable and Linking Format File (`.elf`)	Is the result of compiling a C/C++ application project, that you can download directly to the Nios II processor.
Hexadecimal (Intel-Format) File (`.hex`)	• Contains initialization information for on-chip memories. • The Nios II SBT for Eclipse generate these initialization files for on-chip memories that support initialization of contents.
Flash memory programming data	• Boot code and other arbitrary data you might write to flash memory. • The flash programmer adds appropriate boot code to allow your program to boot from flash memory. • The Nios II SBT for Eclipse includes a flash programmer to allow you to write your program or arbitrary data to flash memory.

This tutorial focuses on downloading only the `.elf` directly to the Nios II system.

3.6.4.5. Running and Debugging Software on the Target Board

The Nios II SBT for Eclipse has the capability to download software to a target board, and run or debug the program on hardware. The Nios II SBT for Eclipse debugger allows you to start and stop the processor, step through code, set breakpoints, and analyze variables as the program executes.

3.6.4.6. Varying the Development Flow

The development flow is not strictly linear. The following lost the common variations:

- Refining the Software and Hardware
- Iteratively Creating a Nios II System
- Verifying the System with Hardware Simulation Tools

Refining the Software and Hardware

After running software on the target board, you might discover that the Nios II system requires higher performance. In this case, you can:

- return to software design steps to make improvements to the software algorithm; or
- return to hardware design steps to add acceleration logic

If the system performs multiple mutually exclusive tasks, you might even decide to use two (or more) Nios II processors that divide the workload and improve the performance of each individual processor.

Iteratively Creating a Nios II System

A common technique for building a complex Nios II system is to start with a simpler Platform Designer system, and iteratively add to it. At each iteration, you can verify that the system performs as expected. You might choose to verify the fundamental components of a system, such as the processor, memory, and communication channels, before adding more complex components. When developing a custom component or a custom instruction, first integrate the custom logic into a minimal system to verify that it works as expected; then integrate the custom logic into a more complex system.

Verifying the System with Hardware Simulation Tools

You can perform hardware simulation of software executing on the Nios II system, using tools such as the ModelSim® RTL simulator. Hardware simulation is useful to meet certain needs, including the following cases:

- To verify the cycle-accurate performance of a Nios II system before target hardware is available.
- To verify the functionality of a custom component or a Nios II custom instruction before trying it on hardware.

If you are building a Nios II system based on the standard components provided with the Nios II EDS, the easiest way to verify functionality is to download the hardware and software directly to a development board.

3.6.5. Creating the Design Example

First, you must install the Intel Quartus Prime software and the Nios II EDS. You must also download tutorial design files from the Intel web site. The design files provide a ready-made Intel Quartus Prime project to use as a starting point.

3.6.5.1. Install the Design Files

Perform the following steps to set up the design environment:

1. Locate the zipped design files on the Intel web site.
2. Unzip the contents of the zip file to a directory on your computer. Do not use spaces in the directory path name.

   The remainder of this tutorial refers to this directory as the `<design files directory>`.

3.6.5.2. Analyze System Requirements

The system requirements are derived from the following goals of the tutorial design example:

- Demonstrate a simple Nios II processor system that you can use for control applications.
- Build a practical, real-world system, while providing an educational experience.
- Demonstrate the most common and effective techniques to build practical, custom Nios II systems.
- Build a Nios II system that works on any board with an Intel FPGA. The entire system must use only on-chip resources, and not rely on the target board.
- The design should conserve on-chip logic and memory resources so it can fit in a wide range of target FPGAs.

These goals lead to the following design decisions:

- The Nios II system uses only the following inputs and outputs:
  - One clock input, which can be any constant frequency.
  - Eight optional outputs to control LEDs on the target board.
- The design uses the following components:
  - Nios II/f core with 2 KB of instruction cache with static branch prediction
  - 20 KB of on-chip memory
  - Timer
  - JTAG UART
  - Eight output-only parallel I/O (PIO) pins
  - System ID component

Related Information

Embedded Peripherals IP User Guide

3.6.5.3. Start the Intel Quartus Prime Software and Open the Example Project

The Intel Quartus Prime project serves as an easy starting point for the Nios II development flow. The Intel Quartus Prime project contains all settings and design files required to create the `.sof`. To open the Intel Quartus Prime project, perform the following steps:

1. Start the Intel Quartus Prime software.
2. Click **Open Existing Project** on the splash screen, or, on the **File** menu, click **Open Project**.
   The **Open Project** dialog box appears.
3. Browse to the `<design files directory>`.
4. Select the file `nios2_quartus2_project.qpf` and click **Open**.
5. To display the Block Diagram File (`.bdf`) `nios2_quartus2_project.bdf`, perform the following steps:
   a. On the **File** menu, click **Open**.
      The **Open** dialog box appears.
   b. Browse to the `<design files directory>`.
   c. Select `nios2_quartus2_project.bdf` and click **Open.**

   The `.bdf` contains an input pin for the clock input and eight output pins to drive LEDs on the board.

Next, you create a new Platform Designer system, which you ultimately connect to these pins.

3.6.5.4. Create a New Platform Designer System

You use Platform Designer to generate the Nios II processor system, adding the desired components, and configuring how they connect together. To create a new Platform Designer system, click **Platform Designer** on the **Tools** menu in the Intel Quartus Prime software. Platform Designer starts and displays the **System Contents** tab.

3.6.5.5. Define the System in Platform Designer

You use Platform Designer to define the hardware characteristics of the Nios II system, such as which Nios II core to use, and what components to include in the system. Platform Designer does not define software behavior, such as where in memory to store instructions or where to send the `stderr` character stream.

The Platform Designer design process does not need to be linear. The design steps in this tutorial are presented in the most straightforward order for a new user to understand. However, you can perform Platform Designer design steps in a different order.

3.6.5.5.1. Specify Target FPGA and Clock Settings

To specify target FPGA and clock settings, perform the following steps:

1. On the **Project Settings** tab, select the **Device Family** that matches the Intel FPGA you are targeting.

If a warning appears stating the selected device family does not match the Quartus project settings, ignore the warning. You specify the device in the Quartus project settings later in this tutorial.

2. In the documentation for your board, look up the clock frequency of the oscillator that drives the FPGA.
3. On the **Clock Settings** tab, double-click the clock frequency in the **MHz** column for `clk_0`. `clk_0` is the default clock input name for the Platform Designer system. The frequency you specify for `clk_0` must match the oscillator that drives the FPGA.
4. Type the clock frequency and press Enter.

Next, you begin to add hardware components to the Platform Designer system. As you add each component, you configure it appropriately to match the design specifications.

Related Information

All Development Kits

3.6.5.5.2. Add the On-Chip Memory

Processor systems require at least one memory for data and instructions. This design example uses one 20 KB on-chip memory for both data and instructions. To add the memory, perform the following steps:

1. On the **IP Catalog** tab (to the left of the **System Contents** tab), expand **Basic Functions**, expand **On-Chip Memory**, and then click **On-Chip Memory (RAM or ROM)**.
2. Click **Add**.
   The On-Chip Memory (RAM or ROM) parameter editor appears.
3. In the **Block type** list, select **Auto**.
4. In the **Total memory size** box, type `20480` to specify a memory size of 20 KB.

   Do not change any of the other default settings.
5. Click **Finish**. You return to Platform Designer.
6. Click the **System Contents** tab.
   An instance of the on-chip memory appears in the system contents table.
7. In the **Name** column of the system contents table, right-click the on-chip memory and click **Rename**.
8. Type `onchip_mem` and press **Enter**.

   You must type these tutorial component names exactly as specified. Otherwise, the tutorial programs written for this Nios II system fail in later steps. In general, it is a good habit to give descriptive names to hardware components. Nios II programs use these symbolic names to access the component hardware. Therefore, your choice of component names can make Nios II programs easier to read and understand.

3.6.5.5.3. Add the Nios II Processor Core

You add the Nios II/f core and configure it to use 2 KB of on-chip instruction cache memory, no data cache and use static branch prediction. For this tutorial, the Nios II/f core is configured to provide a balanced trade-off between performance and resource utilization. To add a Nios II/f core to the system, perform the following steps:

1. On the **IP Catalog** tab, expand **Processors and Peripherals**, and then click **Nios II Gen2 Processor**.
2. Click **Add**.
   The Nios II Processor parameter editor appears, displaying the **Core Nios II** tab.
3. In the **Main Tab** under **Select an Implementation**, select **Nios II/f**.
4. Click **Finish** and return to the **Platform Designer System Contents** tab.
   The Nios II core instance appears in the system contents table. Ignore the exception and reset vector error messages. You resolve these errors in future steps.
5. In the **Name** column, right-click the **Nios II** processor and click **Rename**.
6. Type `cpu` and press **Enter**.
7. In the **Connections** column, connect the `clk` port of the **clk_0** clock source to both the `clk1` port of the on-chip memory and the clk port of the Nios II processor by clicking the hollow dots on the connection line. The dots become solid indicating the ports are connected.
8. Connect the `clk_reset` port of the **clk_0** clock source to both the `reset1` port of the on-chip memory and the `reset_n` port of the Nios II processor.
9. Connect the `s1` port of the on-chip memory to both the `data_master` port and `instruction_master` port of the Nios II processor.
10. Double-click the Nios II processor row of the system contents table to reopen the Nios II Processor parameter editor.
11. Under **Reset Vector** in **Vectors** tab, select **onchip_mem.s1** in the **Reset vector memory** list and type `0x0` in the **Reset vector offset** box.
12. Under **Exception Vector**, select **onchip_mem.s1** in the **Exception vector memory** list and type `0x20` in the **Exception vector offset** box.
13. Click the Caches and Memory Interfaces tab.
14. In the Instruction cache list, select 2 Kbytes.
15. Choose **None** for **Data Cache size** and do not change other default settings.
16. In **Advanced Features** tab, select **Static branch prediction** type.
17. Click **Finish**. You return to the Platform Designer System Contents tab.

    Do not change any settings on the **MMU and MPU Settings** and **JTAG Debug** tabs.

3.6.5.5.4. Add the JTAG UART

The JTAG UART provides a convenient way to communicate character data with the Nios II processor through the Intel FPGA Download Cable. To add the JTAG UART, perform the following steps:

1. On the **IP Catalog** tab, expand **Interface Protocols**, expand **Serial**, and then click **JTAG UART**.
2. Click **Add**.
   The **JTAG UART** parameter editor appears and do not change the default settings.
3. Click **Finish** and return to the **Platform Designer System Contents** tab.
   The JTAG UART instance appears in the system contents table.
4. In the **Name** column, right-click the **JTAG UART** and click **Rename**.
5. Type `jtag_uart` and press **Enter**.
6. Connect the `clk` port of the **clk_0** clock source to the `clk` port of the **JTAG UART**.
7. Connect the `clk_reset` port of the **clk_0** clock source to the `reset` port of the **JTAG UART**.
8. Connect the `data_master` port of the Nios II processor to the `avalan_jtag_slave` port of the **JTAG UART**.

   The `instruction_master` port of the Nios II processor does not connect to the JTAG UART because the JTAG UART is not a memory device and cannot send instructions to the Nios II processor.

Related Information

Embedded Peripherals IP User Guide

3.6.5.5.5. Add the Interval Timer

Most control systems use a timer component to enable precise calculation of time. To provide a periodic system clock tick, the Nios II HAL requires a timer. To add the timer, perform the following steps:

1. On the **IP Catalog** tab, expand **Processors and Peripherals**, expand **Peripherals**, and then click **Interval Timer**.
2. Click **Add**.
   The **Interval Timer** parameter editor appears.
3. Click **Finish** return to the **Platform Designer System Contents** tab.
   The interval timer instance appears in the system contents table.
4. In the **Name** column, right-click the **interval timer** and click **Rename.**
5. Type `sys_clk_timer` and press **Enter.**
6. Connect the `clk` port of the **clk_0** clock source to the `clk` port of the **interval timer**.
7. Connect the `clk_reset` port of the **clk_0** clock source to the `reset` port of the **interval timer**.
8. Connect the `data_master port` of the **Nios II processor** to the `s1` port of the **interval timer**.

Related Information

Embedded Peripherals IP User Guide

3.6.5.5.6. Add the System ID Peripheral

The system ID peripheral safeguards against accidentally downloading software compiled for a different Nios II system. If the system includes the system ID peripheral, the Nios II SBT for Eclipse can prevent you from downloading programs compiled for a different system. To add system ID peripheral, perform the following steps:

1. On the **IP Catalog** tab, expand **Basic Functions**, expand **Simulations; Debug and Verifications** and then click **System ID Peripheral**.
2. Click **Add**.
   The **System ID Peripheral** parameter editor appears and do not change the default setting.
3. Click **Finish** and return to the **Platform Designer System Contents** tab.
   The system ID peripheral instance appears in the system contents table.
4. In the **Name** column, right-click the **system ID peripheral** and click **Rename.**
5. Type `sysid` and press **Enter**.
6. Connect the `clk` port of the **clk_0** clock source to the `clk` port of the **system ID peripheral**.
7. Connect the `clk_reset` port of the **clk_0** clock source to the `reset` port of the **system ID peripheral**.
8. Connect the `data_master` port of the **Nios II processor** to the `control_slave` port of the **system ID peripheral**.

Related Information

Embedded Peripherals IP User Guide

3.6.5.5.7. Add the PIO

PIO signals provide an easy method for Nios II processor systems to receive input stimuli and drive output signals. Complex control applications might use hundreds of PIO signals which the Nios II processor can monitor. This design example uses eight PIO signals to drive LEDs on the board. To add the PIO, perform the following steps:

Note: Perform these steps even if your target board does not have LEDs.

1. On the **IP Catalog** tab, expand **Processors and Peripherals**, expand **Peripherals**, and then click **PIO** .
2. Click **Add**.
   The **PIO (Parallel I/O)** parameter editor appears and do not change the default settings.
3. Click **Finish** and return to the **Platform Designer System Contents** tab.
   The PIO instance appears in the system contents table.
4. In the **Name** column, right-click the **PIO** and click **Rename**.
5. Type `led_pio` and press **Enter**.
6. Connect the `clk` port of the **clk_0** clock source to the `clk` port of the **PIO**.

7. Connect the `clk_reset` port of the **clk_0** clock source to the `reset` port of the **PIO**.
8. Connect the `data_master` port of the **Nios II processor** to the `s1` port of the **PIO**.
9. In the **external_connection** row, click **Click to export** in the **Export** column to export the PIO ports.

Related Information

Embedded Peripherals IP User Guide

3.6.5.5.8. Specify Base Addresses and Interrupt Request Priorities

To specify how the components added in the design to interact to form a system, you need assign base addresses for each slave component, and assign interrupt request (IRQ) priorities for the JTAG UART and the interval timer.

Platform Designer provides the Assign Base Addresses command which makes assigning component base addresses easy. For many systems, including this design example, Assign Base Addresses is adequate. However, you can adjust the base addresses to suit your needs. Below are some guidelines for assigning base addresses:

- Nios II processor cores can address a 31-bit address span. You must assign base address between `0x00000000` and `0x7FFFFFFF`.

  Note: The **Use most-significant address bit in processor to bypass data cache** option is enable by default. If disabled, the Nios II processor cores supports full 32-bit address.
- Nios II programs use symbolic constants to refer to addresses. You do not have to choose address values that are easy to remember.
- Address values that differentiate components with only a one-bit address difference produce more efficient hardware. You do not have to compact all base addresses into the smallest possible address range, because this can create less efficient hardware.
- Platform Designer does not attempt to align separate memory components in a contiguous memory range. For example, if you want an on-chip RAM and an off-chip RAM to be addressable as one contiguous memory range, you must explicitly assign base addresses.

Platform Designer also provides an **Assign Interrupt Numbers** command which connects IRQ signals to produce valid hardware results. However, assigning IRQs effectively requires an understanding of how software responds to them. Because Platform Designer does not know the software behavior, Platform Designer cannot make educated guesses about the best IRQ assignment.

The Nios II HAL interprets low IRQ values as higher priority. The timer component must have the highest IRQ priority to maintain the accuracy of the system clock tick.

To assign appropriate base addresses and IRQs, perform the following steps:

1. On the **System** menu, click **Assign Base Addresses** to make Platform Designer assign functional base addresses to each component in the system. Values in the **Base** and **End** columns might change, reflecting the addresses that Platform Designer reassigned.
2. In the **IRQ** column, connect the Nios II processor to the JTAG UART and interval timer.
3. Click the **IRQ** value for the **jtag_uart** component to select it.
4. Type 16 and press **Enter** to assign a new IRQ value.
5. Click the **IRQ value** for the **sys_clk_timer** component to select it.
6. Type 1 and press **Enter** to assign a new IRQ value.

3.6.5.5.9. Generate the Platform Designer System

To generate the Platform Designer system, perform the following steps:

1. Click the **Generation** tab.
2. Select **None** in both the **Create simulation model** and **Create testbench Platform Designer system** lists.

   Because this tutorial does not cover the hardware simulation flow, you can select these settings to shorten generation time.
3. Click **Generate**. Click **Yes** when the **Save changes?** dialog box appears.
4. Type `first_nios2_system` in the **File name** box and click **Save**.
   The **Generate** dialog box appears and system generation process begins. The generation process can take several minutes. When generation completes, Platform Designer prompt: `Create HDL design files for synthesis`.
5. Click **Close** to close the dialog box.
6. On the **File** menu, click **Exit** to close Platform Designer and return to the Intel Quartus Prime software.

You are ready to integrate the system into the Intel Quartus Prime hardware project and use the Nios II SBT for Eclipse to develop software.

3.6.5.6. Integrate the Platform Designer System into the Intel Quartus Prime Project

To complete the hardware design, you need to perform the following tasks:

- Instantiate the Platform Designer system module in the Intel Quartus Prime project.
- Assign FPGA device and pin locations.
- Compile the Intel Quartus Prime project.
- Verify timing.

3.6.5.6.1. Instantiate the Platform Designer System Module in the Intel Quartus Prime Project

Platform Designer outputs a design entity called the system module. The tutorial design example uses the block diagram method of design entry, so you instantiate a system module symbol **first_nios2_system** into the `.bdf`.

Note: How you instantiate the system module depends on the design entry method of the overall Intel Quartus Prime project. For example, if you were using Verilog HDL for design entry, you would instantiate the Verilog module `first_nios2_system` defined in the file `first_nios2_system.v`.

To instantiate the system module in the `.bdf`, perform the following steps:

1. Double-click in the empty space to the right of the input and output wires. The **Symbol** dialog box appears.
2. Under **Libraries**, expand **Project**.
3. Click **first_nios2_system**. The **Symbol** dialog box displays the **first_nios2_system** symbol.
4. Click **OK**. You return to the `.bdf` schematic. The **first_nios2_system** symbol tracks with your mouse pointer.
5. Position the symbol so the pins on the symbol align with the wires on the schematic.
6. Click to anchor the symbol in place.
7. If your target board does not have LEDs that the Nios II system can drive, you must delete the **LEDG[7..0]** pins. To delete the pins, perform the following steps:
   a. Click the output symbol **LEDG[7..0]** to select it.
   b. On your keyboard, press **Delete.**
8. To save the completed `.bdf`, click **Save** on the **File** menu.

3.6.5.6.2. Add IP Variation File

You can add the Intel Quartus Prime IP File (`.qip`) to the your Intel Quartus Prime project by performing the following steps:

1. On the **Assignments** menu, click **Settings**. The **Settings** dialog box appears.
2. Under **Category**, click **Files**. The **Files** page appears.
3. Next to File name, click the **browse (...)** button.
4. In the **Files** of type list, select **Script Files (*.tcl, *.sdc, *.qip)**.
5. Browse to locate `<design files directory>/first_nios2_system/synthesis/ first_nios2_system.qip` and click **Open** to select the file.
6. Click **Add** to include `first_nios2_system.qip` in the project.
7. Click **OK** to close the **Settings** dialog box.

3.6.5.6.3. Assign FPGA Device

Before assigning FPGA pin locations to match the pinouts of your board, you need to first assign a specific target device. To assign the device, perform the following steps:

1. On the **Assignments** menu, click **Device**.
   The **Device** dialog box appears.
2. In the **Family** list, select the FPGA family that matches your board.

   If prompted to remove location assignments, do so.
3. Under **Target device**, select **Specific device selected in Available devices list**.
4. Under **Available devices**, select the exact device that matches your board.

   If prompted to remove location assignments, do so.
5. Click **OK** to accept the device assignment.

3.6.5.6.4. Assign FPGA Pin Locations

Before assigning the FPGA pins, you must know the pin layout for the board. You also must know other requirements for using the board, refer to related information below. To assign the FPGA pin locations, perform the following steps:

1. On the Processing menu, point to **Start**, and click **Start Analysis & Elaboration** to prepare for assigning pin locations.
   The analysis starts by displaying a `data not available` message and can take several minutes. A confirmation message box appears when analysis and elaboration completes.
2. Click **OK.**
3. On the **Assignments** menu, click **Pin Planner**.
   The Intel Quartus Prime **Pin Planner** appears.
4. In the **Node Name** column, locate **PLD_CLOCKINPUT**.
5. In the **PLD_CLOCKINPUT** row, double-click in the **Location** cell to access a list of available pin locations.
6. Select the appropriate FPGA pin that connects to the oscillator on the board.

   If your design fails to work, recheck your board documentation for this step first.
7. In the **PLD_CLOCKINPUT** row, double-click in the **I/O Standard** cell to access a list of available I/O standards.
8. Select the appropriate I/O standard that connects to the oscillator on the board.
9. If you connected the LED pins in the board design schematic, repeat steps 4 to 8 for each of the LED output pins (**LEDG[0]**, **LEDG[1]**, **LEDG[2]**, **LEDG[3]**, **LEDG[4]**, **LEDG[5]**, **LEDG[6]**, **LEDG[7]**) to assign appropriate pin locations.
10. On the **File** menu, click **Close** to save the assignments.
11. On the **Assignments** menu, click **Device**.
    The **Device** dialog box appears.
12. Click **Device and Pin Options**.
    The **Device and Pin Options** dialog box appears.
13. Click the **Unused Pins** page.

14. In the **Reserve all unused pins** list, select **As input tri-stated with weak pull-up**. With this setting, all unused I/O pins on the FPGA enter a high-impedance state after power-up.

    Unused pins are set as input tri-stated with weak pull-up to remove contention which might damage the board. Depending on the board, you might have to make more assignments for the project to function correctly. You can damage the board if you fail to account for the board design. Consult with the maker of the board for specific contention information.
15. Click **OK** to close the **Device and Pin Options** dialog box.
16. Click **OK** to close the **Device** dialog box.

3.6.5.6.5. Set Timing

To ensure the design meets timing, perform the following steps:

1. On the **File** menu, click **Open**.
2. In the **Files of type** list, select **Script Files (*.tcl, *.sdc, *.qip)**.
3. Browse to locate `<design files directory>/hw_dev_tutorial.sdc` and click **Open**. The file opens in the text editor.
4. Locate the following `create_clock` command:`create_clock -name sopc_clk -period 20 [get_ports PLD_CLOCKINPUT]`
5. Change the period setting from 20 to the clock period (1/frequency) in nanoseconds of the oscillator driving the clock pin.
6. On the **File** menu, click **Save.**
7. On the **Assignments** menu, click **Settings**.
   The **Settings** dialog box appears.
8. Under **Category**, click **TimeQuest Timing Analyzer**.
9. Next to File name, click the browse **(...)** button.
10. Browse to locate `<design files directory>/hw_dev_tutorial.sdc` and click **Open** to select the file.
11. Click **Add** to include `hw_dev_tutorial.sdc` in the project.
12. Turn on **Enable multicorner timing analysis during compilation**.
13. Click **OK**.

3.6.5.6.6. Compile the Intel Quartus Prime Project and Verify Timing

To create a `.sof` file, you have to compile the hardware design and then it download to the board. After the compilation completes, you must analyze the timing performance of the FPGA design to verify that the design works in hardware. To compile the Intel Quartus Prime project, perform the following steps:

1. On the **Processing** menu, click **Start Compilation**.

The **Tasks** window and percentage and time counters in the lower-right corner display progress. The compilation process can take several minutes. When compilation completes, a dialog box displays the message "Full Compilation was successful."

2. Click **OK**. The Quartus Prime software displays the **Compilation Report** tab.
3. Expand the **TimeQuest Timing Analyzer** category in the compilation report.
4. Click **Multicorner Timing Analysis Summary**.
5. Verify that the **Worst-case Slack** values are positive numbers for **Setup**, **Hold**, **Recovery**, and **Removal**.

   If any of these values are negative, the design might not operate properly in hardware. To meet timing, adjust Intel Quartus Prime assignments to optimize fitting, or reduce the oscillator frequency driving the FPGA.

3.6.5.7. Download the Hardware Design to the Target FPGA

To download the `.sof` to the target board, perform the following steps:

1. Connect the board to the host computer with the download cable, and apply power to the board.
2. On the **Tools** menu in the Intel Quartus Prime software, click **Programmer**.
   The Intel Quartus Prime Programmer appears and automatically displays the appropriate configuration file (`nios2_quartus2_project.sof`).
3. Click **Hardware Setup** in the upper left corner of the Intel Quartus Prime Programmer to verify your download cable settings.
   The **Hardware Setup** dialog box appears.
4. Select the appropriate download cable in the **Currently selected hardware** list.

   If the appropriate download cable does not appear in the list, you must first install drivers for the cable.
5. Click **Close**.
6. In the **nios2_quartus2_project.sof** row, turn on **Program/Configure**.
7. Click **Start**.
   The Progress meter sweeps to 100% as the Intel Quartus Prime software configures the FPGA.

At this point, the Nios II system is configured and running in the FPGA, but it does not yet have a program in memory to execute.

3.6.5.8. Develop Software Using the Nios II SBT for Eclipse

Developing software using the Nios II SBT for Eclipse consists the following tasks:

- Create new Nios II C/C++ application and BSP projects.
- Compile the projects.

To perform these steps, you must have the `.sopcinfo` file you created earlier in this tutorial. Refer to related information for more information.

Note: This tutorial presents only the most basic software development steps to demonstrate software running on the hardware system you created in previous sections.

Related Information

Define the System in Platform Designer on page 76

3.6.5.8.1. Create a New Nios II Application and BSP from Template

To create new Nios II C/C++ application and BSP projects, perform the following steps:

1. Start the Nios II SBT for Eclipse. On Windows computers, click **Start**, point to **Programs**, **Nios II EDS <version>**, and then click **Nios II <version> Software Build Tools for Eclipse**. On Linux computers, run the executable file `<Nios II EDS install path>/bin/eclipse-nios2`.
2. If the **Workspace Launcher** dialog box appears, click **OK** to accept the default workspace location.
3. On the **Window** menu, point to **Open Perspective**, and then either click **Nios II**, or click **Other** and then click **Nios II** to ensure you are using the Nios II perspective.
4. On the **File** menu, point to **New**, and then click **Nios II Application and BSP from Template**.
   The **Nios II Application and BSP from Template** wizard appears.
5. Under **Target hardware information**, next to **SOPC Information File name**, browse to locate the `<design files directory>`.
6. Select `first_nios2_system.sopcinfo` and click **Open**.
   Nios II Application and BSP from Template wizard shows the current information for the **SOPC Information File name** and **CPU name** fields.
7. In the **Project name** box, type `count_binary`.
8. In the **Templates** list, select **Count Binary**
9. Click **Finish**.

The Nios II SBT for Eclipse creates and displays the following new projects in the Project Explorer view, typically on the left side of the window:

- `count_binary`—Your C/C++ application project
- `count_binary_bsp`—A board support package that encapsulates the details of the Nios II system hardware

3.6.5.8.2. Compile the Project

You have to compile the project to produce an executable software image. For the tutorial design example, you must first adjust the project settings to minimize the memory footprint of the software, because your Nios II hardware system contains only 20 KB of memory. To adjust the project settings and compile the project, perform the following steps:

1. In the **Project Explorer** view, right-click **count_binary_bsp** and click **Properties**. The **Properties for count_binary_bsp** dialog box appears.
2. Click the **Nios II BSP Properties** page. The Nios II BSP Properties page contains basic software build settings.

Though not needed for this tutorial, note the BSP Editor button in the lower right corner of the dialog box. You use the Nios II BSP Editor to access advanced BSP settings.

3. Adjust the following settings to reduce the size of the compiled executable:
   a. Turn on **enable_reduced_device_drivers**.
   b. Turn off **enable_gprof**.
   c. Turn on **enable_small_c_library**.
   d. Turn off **enable_sim_optimize**.
4. Click **OK**.
   The BSP regenerates, the **Properties** dialog box closes, and you return to the Nios II SBT for Eclipse.
5. In the **Project Explorer** view, right-click the **count_binary** project and click **Build Project**.

The **Build Project** dialog box appears, and the Nios II SBT for Eclipse begins compiling the project. When compilation completes, a `count_binary build complete` message appears in the Console view.

3.6.5.9. Run the Program on Target Hardware

To download the software executable to the target board, perform the following steps:

1. Right-click the **count_binary** project, point to **Run As**, and then click **Nios II Hardware**.

   If the **Run Configurations** dialog box appears, verify that **Project name** and **ELF file name** contain relevant data, then click **Run**.

   The Nios II SBT for Eclipse downloads the program to the FPGA on the target board and the program starts running. When the target hardware starts running the program, the Nios II Console view displays character I/O output. If you connected LEDs to the Nios II system in previous section, then the LEDs blink in a binary counting pattern.
2. Click the **Terminate** icon (the red square) on the tool bar of the Nios II Console view to terminate the run session and the Nios II SBT for Eclipse disconnects from the target hardware.

   You can edit the `count_binary.c` program in the Nios II SBT for Eclipse text editor and repeat these two steps to witness your changes executing on the target board. If you rerun the program, buffered characters from the previous run session might display in the Console view before the program begins executing.

Related Information

Integrate the Platform Designer System into the Intel Quartus Prime Project on page 82

3.7. Platform Designer System Design Tutorial

This tutorial introduces you to the Platform Designer system integration tool available with the Intel Quartus Prime software.

This tutorial shows you how to design a system that uses various test patterns to test an external memory device. It guides you through system requirement analysis, hardware design tasks, and evaluation of the system performance, with emphasis on system architecture.

In this tutorial, you create a memory tester system that tests a synchronous dynamic random access memory (SDRAM) device. The final system contains the SDRAM controller and instantiates a Nios® II processor and embedded peripherals in a hierarchical subsystem. The final design includes various Platform Designer components that generate test data, access memory, and verify the returned data.

The memory tester components for the design are Verilog HDL components with an accompanying Hardware Component Description File (**_hw.tcl**) that describes the interfaces and parameterization of each component. The **_hw.tcl** files are located in the `tt_qsys_design\memory_tester_ip` directory.

The final system contains the following components:

- Processor subsystem based on the Nios II/e core, which includes an on-chip RAM to store the processor's software code, and a JTAG UART to communicate via JTAG and display the memory test results in the host PC's console.
- SDRAM controller to control the off-chip DDR SDRAM device under test.
- Custom and pseudo-random binary sequence (PRBS) pattern generators and checkers to test the robustness of links.
- Pattern select multiplexer and demultiplexer to choose between the two pattern generators and checkers.
- Pattern writer and reader that interact with the SDRAM controller.
- Memory test controller.

Each section in this tutorial provides an overview describing the components that you instantiate. You can use the final system on hardware without a license, and perform the following actions with Intel FPGA IP Evaluation Mode feature:

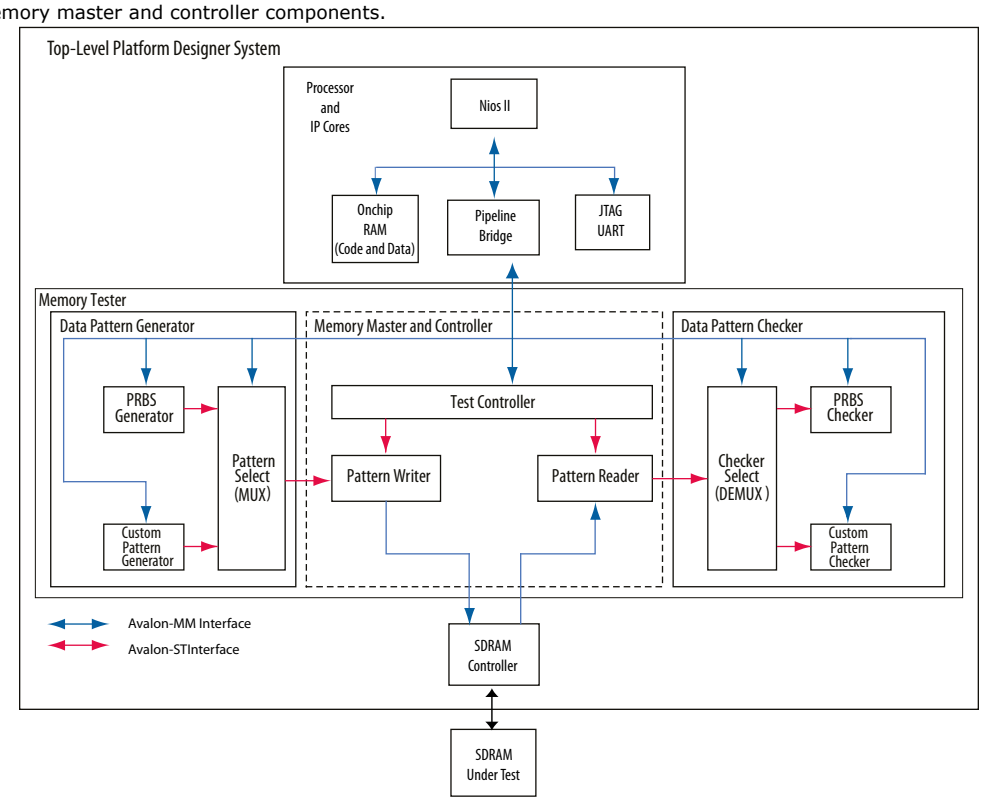
- Simulate the behavior of the system and verify its functionality.
- Generate time-limited device programming files for designs that incorporate Intel or partner IP.
- Program a device and verify your design in hardware.

You can use the Nios II/e processor and the DDR SDRAM IP cores with a Quartus Prime subscription license. Design files for other development kit boards use different DDR SDRAM controllers to match the memory device available on the development kit.

In this tutorial, you instantiate the complete memory tester system in the top-level system along with the processor IP Cores, which are grouped as their own processor system, and the SDRAM Controller IP. The Nios II processor includes a software program to control the memory tester system, which communicates with the SDRAM Controller to access the off-chip SDRAM device under test.

Figure 28. Platform Designer Memory Tester

The components in the memory tester system are grouped into a single Platform Designer system with three major design functions. The design hierarchy allows you to instantiate the data pattern generator and data pattern checker components into separate systems. You can then add the memory tester system with the memory master and controller components.



Related Information

AN320: Using Intel FPGA IP Evaluation Mode

3.7.1. Software and Hardware Requirements

The Platform Designer System Design tutorial requires the following software and hardware requirements:

- Intel Quartus Prime software.
- Nios II EDS.
- **tt_qsys_design.zip** design files, available from the **Platform Designer Tutorial Design Example** page. The design files include project files set up for select Intel development boards, and components that you can use in any Platform Designer design.

You can build the Platform Designer system in this tutorial for any Intel development board or your own custom board, if it meets the following requirements:

- An Intel Arria, Cyclone, or Stratix series FPGA.
- Minimum of 12k logic elements (LEs).
- Minimum of 128k of embedded memory.
- JTAG connection to the FPGA that provides a communications link back to the host so that you can monitor the memory test progress.
- Any memory that has a Platform Designer-based controller with an Avalon® Memory-Mapped (Avalon-MM) slave interface.

Related Information

- Intel FPGA Software Installation and Licensing Manual.
- Platform Designer Tutorial Design Example

3.7.2. Download and Install the Tutorial Design Files

1. On the **Platform Designer Tutorial Design Example** page, under **Using this Design Example**, click **Platform Designer Tutorial Design Example (.zip)** to download and install the tutorial design files for the Platform Designer tutorial.
2. Extract the contents of the archive file to a directory on your computer. Do not use spaces in the directory path name.

In place of following all steps in this tutorial to create subsystem, hierarchical, and top-level design files, you can copy the completed design files listed below into the `tt_qsys_design\quartus_ii_projects_for_boards\<development_board_type>` directory, depending on your board type.

- The two completed subsystems: **pattern_generator_system.qsys**, and **pattern_checker_system.qsys** from the `tt_qsys_design\completed_subsystems` directory.
- The hierarchical system **memory_tester_system.qsys** from the **tt_qsys_design\completed_subsystems\completed_memory_tester_system** directory.
- The top-level hierarchical system **top_system.qsys** from the **tt_qsys_design\quartus_ii_projects_for_boards\<development_board_type>\backup_and_completed_top_system\completed_top_system** directory.

Related Information

- Platform Designer Tutorial Design Example
- Platform Designer Tutorial Design Example (.zip)

3.7.3. Open the Tutorial Project

The design files for the Platform Designer tutorial provide the custom IP design blocks that you need, and a partially completed Intel Quartus Prime project and Platform Designer system.

The following design requirements are included in the Platform Designer tutorial design files:

- Intel Quartus Prime project I/O pin assignments and Synopsys Design Constraint (.sdc) timing assignments for each supported development board.
- Parameterized Nios II processor core and software to communicate with the host PC that controls the memory test system that you develop.
- Parameterized DDR SDRAM controller to use the memory on the development board.

To open the tutorial project:

1. Open the Intel Quartus Prime software.
2. To open the Intel Quartus Prime Project File (**.qpf**) for your board, click **File ➤ Open Project**.
3. Browse to the `tt_qsys_design\quartus_ii_projects_for_boards\<development_board>\` directory.
4. Select the relevant board-specific **.qpf** file, and then click **Open**.

3.7.4. Creating Platform Designer Systems

The data pattern generator and data pattern checker are design blocks for the memory tester system. In this tutorial, you learn to instantiate, parameterize, and connect components by creating the data pattern generator and data pattern checker Platform Designer systems.

- **Data pattern generator**—The data pattern generator generates high-speed streaming data, which performs either as a PRBS, or as a soft programmable sequence, for example, "walking ones." The design sends the data with an Avalon-Streaming (Avalon-ST) connection to the pattern writer of the memory master and control logic. The data pattern generator writes the data to memory based on commands issued by the controller logic. When the design writes the data to memory, the pattern reader logic reads the contents back and sends it to the data pattern verification logic.
- **Data pattern checker**—The data pattern checker accepts the data read back by the pattern reader from an Avalon-ST connection. The design verifies the data pattern to ensure that the pattern it writes to memory is identical to the data that it reads back.

3.7.4.1. Create a Data Pattern Generator Platform Designer System

The data pattern generator includes two components to generate test patterns, and a third component to multiplex the data that a processor controls. You configure the pattern generator to match the width of the memory interface. Because the data pattern generator provides a full word of data every clock cycle, configuring the components to match the memory width provides sufficient bandwidth to access the memory.

Note: As you add components and make connections in your Platform Designer system, error and warning messages appear in the Platform Designer **Messages** tab, indicating steps that you must perform before the system is complete. Some error messages appear between steps and are not resolved immediately; as you progress through the tutorial, errors are resolved, and the error messages disappear.

You must use the exact system names described in this tutorial in order for the provided scripts to function correctly.

3.7.4.1.1. Create a New Platform Designer System and Set up the Clock Source

1. In the Intel Quartus Prime software, click **Tools ➤ Platform Designer** to create a new Platform Designer design.
2. In the **System Contents** tab, Platform Designer shows a clock source instance, **clk_0**. To open the clock source settings, right-click **clk_0**, and then click **Edit**.
3. Turn off **Clock frequency is known** to indicate that, when created, the higher-level hierarchical system that instantiates this subsystem provides the clock frequency.
4. Click **Finish**.
5. Click **File ➤ Save As** to save the Platform Designer system.
6. In the **Save As** dialog box, type **pattern_generator_system**, and then click **Save**.

If Platform Designer prompts you to open the **top_system.qsys** file, click **Cancel** in the **Open** dialog box

3.7.4.1.2. Add a Pipeline Bridge

The components that make up the data pattern generator include several Avalon-MM slave interfaces. To allow a higher-level system to access the Avalon-MM slave interfaces by reading and writing to a single slave interface, you can consolidate the slave interfaces behind an Avalon-MM pipeline bridge, and export a single Avalon-MM slave interface out of the system.

To determine the required address width for a bridge, you must know the required addresses span of the other components in the system. Memory-mapped component interfaces outside the system address each interface in the system by specifying a memory offset value relative to the base address of the bridge.

A pipeline bridge can also improve system timing performance by optionally adding pipeline registers to the design.

1. In the Library search box, type **bridge** to filter the component list and show only bridge components.
2. Select **Avalon-MM Pipeline Bridge**, and then click **Add**.
3. In the parameter editor, under Parameters, type **11** for the **Address width**.
   This width accommodates the memory span of all memory-mapped components behind the bridge in this system. As you add the other components in the system, you specify their base addresses within the span of the address space.
4. Accept all other default settings, and then click **Finish**.
   The pipeline bridge is added to your system with the instance name **mm_bridge_0**.

5. On the **System Contents** tab, right-click **mm_bridge_0**, click **Rename**, and then type **mm_bridge**.
6. In the **Clock** column for the **mm_bridge clk** interface, select **clk_0** from the list.
7. To export the **mm_bridge s0** interface, double-click the **Export** column, and then type **slave**.

3.7.4.1.3. Add a Custom Pattern Generator

The pattern generator generates multiple test patterns to test the off-chip SDRAM device. The custom pattern generator system provides a stream of pattern data via an Avalon-ST source interface.

The component is programmed with the pattern data and a pattern length. When the end of the pattern is reached, the custom pattern generator cycles back to the first element of the pattern. This custom pattern generator generates the following standard memory tester patterns:

- Walking ones
- Walking zeros
- Low frequency
- Alternating low frequency
- High frequency
- Alternating high frequency
- Synchronous PRBS

The width of the memory dictates the walking ones or zeros pattern lengths. For example, when testing a 32-bit memory, the walking ones or zeros pattern is 32 elements in length before repeating. The high and low frequency patterns contain only two elements before repeating. The synchronous PRBS pattern is the longest pattern containing 256 elements before repeating.

This custom pattern generator contains three interfaces, two of which control the generated pattern, and a third interface which control the behavior of the custom pattern generated. The processor accesses the pattern_access interface, which is write only, to program the elements of the custom pattern that are sent to the pattern writer core, and the csr interface, which is used for the control and status registers. The st_pattern_output is the streaming source interface that sends data to the pattern writer core.

To add the custom pattern generator:

1. In the IP Catalog, expand **Memory Test Microcores**, and then double-click **Custom Pattern Generator**.
2. In the parameter editor, accept the default parameters, and then click **Finish**.
3. Rename the instance to **custom_pattern_generator**.
4. Set the **custom_pattern_generator clock** interface to **clk_0**.

5. To connect the **custom_pattern_generator csr** interface to the **mm_bridge m0** interface, in the **Connections** column, click to fill in the connection dot between the **custom_pattern_generator csr** interface and the **mm_bridge m0** interface.
6. Connect the **custom_pattern_generator pattern_access** interface to the **mm_bridge m0** interface.
   The processor accesses the system through the m0 interface to communicate with the csr and pattern_access interfaces.
7. To assign the **custom_pattern_generator csr** interface to a base address of `0400`, in the **Base** column, double-click the `0x00000000` address, and then enter `400` for the base address, which is in hexadecimal format.
   If the Base column is locked for the **custom_pattern_generator csr**, right-click and then click **Unlock Base Address**.

The address space represents memory accessible by the processor. Each address specifies a location in memory that can be addressed and accessed, and each interface must have a unique address range. The address space of each interface is determined by its base address and its memory span, or how much memory is required for that interface.

You can see the default address range of the pattern_access interface in the **Base** and **End** columns on the **System Contents** tab.

You assign a base address for the **csr** interface that is higher than the end address of the **pattern_access** interface to avoid conflicting with the address space of the **pattern_access** interface.

3.7.4.1.4. Add a PRBS Pattern Generator

The output of the PRBS pattern generator is a statically-defined PRBS pattern. You can specify the pattern length before the pattern repeats in the parameter editor. The pattern length is defined by 2^(data width) – 1.

For example, a 32-bit PRBS pattern generator repeats the pattern after it sends 4,294,967,295 elements. You set the width of the PRBS generator based on the (local) data width of the memory on your board.

The PRBS pattern generator has two interfaces; the csr and the st_pattern_output streaming source interface. The csr interface controls the behavior of the PRBS pattern generated. The st_pattern_output streaming source interface sends data to the pattern writer component.

1. In the IP Catalog, expand **Memory Test Microcores**, and then double-click **PRBS Pattern Generator**.
2. In the parameter editor, accept the default parameters, and then click **Finish**.
3. Rename the instance to **prbs_pattern_generator**.
4. Set the **prbs_pattern_generator clock** interface to **clk_0**.
5. Connect the **prbs_pattern_generator csr** interface to the **mm_bridge m0** interface.
6. Assign the **prbs_pattern_generator csr** interface to a base address of **0x0420**, which is a base address just higher than the end address of the **custom_pattern_generator csr** interface of **0x410**.

3.7.4.1.5. Add a Two-to-One Streaming Multiplexer

You add a two-to-one streaming multiplexer between the pattern generators and the pattern writer because the system has two pattern sources, and the pattern writer component accepts data only from one streaming source. The two-to-one streaming soft programmable multiplexer IP core allows the processor to select which pattern to send to the pattern writer.

The two-to-one streaming multiplexer component has the following interfaces:

- Two streaming inputs: **st_input_A** and **st_input_B**.
- One streaming output: **st_output**.
- One csr slave interface, which the processor controls to select whether **input A** or **input B** is sent to the streaming output.

The custom pattern generator connects to input A, and the PRBS pattern generator connects to input B.

1. In the IP Catalog, expand **Memory Test Microcores**, and then double-click **Two-to-one Streaming Mux**.
2. In the parameter editor, accept the default parameters, and then click Finish.
3. Rename the instance to **two_to_one_st_mux**.
4. Set the **two_to_one_st_mux clock** to **clk_0**.
5. Connect the **two_to_one_st_mux st_input_A** interface to the **custom_pattern_generator st_pattern_output** interface.
6. Connect the **two_to_one_st_mux st_input_B** interface to the **prbs_pattern_generator st_pattern_output** interface.
7. Connect the **two_to_one_st_mux csr** interface to the **mm_bridge m0** interface.
8. Export the **two_to_one_st_mux st_output** interface with the name **st_data_out**.
9. Assign the **two_to_one_st_mux csr** interface to a base address of **0x0440**, which is a base address higher than the end address of the **prbs_pattern_generator csr** interface at base address **0x0420**

The output of the two-to-one streaming multiplexer carries the pattern data from either the custom pattern generator or the PRBS pattern generator, to the pattern writer. The data, from the output of the two-to-one streaming multiplexer, achieves a throughput of one word per clock cycle.

3.7.4.1.6. Verify the Memory Address Map

You control the system by accessing the memory locations allocated to each component within the subsystem. To ensure that the memory map of the system you create matches the memory map of other components, you must verify the base addresses for the data pattern generator system.

On the **Address Map** tab, verify that the entries in the **Address Map** table match the values in in the table below. Red exclamation marks indicate that the address ranges overlap. Correct the base addresses, as appropriate, to ensure there are no overlapping addresses, and your map matches this tutorial's guidelines.

Table 9. Address Map Table

Component	Address
custom_pattern_generator.csr	0x00000400 – 0x0000040f
custom_pattern_generator.pattern_access	0x00000000 – 0x000003ff
prbs_pattern_generator.csr	0x00000420 – 0x0000043f
two_to_one_st_mux.csr	0x00000440 – 0x00000447

3.7.4.1.7. Connect the Reset Signals

You must connect all the reset signals, which eliminates the error messages in the **Messages** tab. Platform Designer allows multiple reset domains, or one reset signal for the system. In the design, you want to connect all the reset signals with the incoming reset signal. To connect all the reset signals, on the System menu, select **Create Global Reset Network**.

At this point in the system design, Platform Designer shows no remaining error messages. If you have any error messages in the **Messages** tab, review the procedures to create this system to ensure you did not miss a step. You can view the reset connections and the timing adapters on the **System Contents** tab, and by selecting **Show System With Platform Designer Interconnect** on the System menu.

3.7.4.1.8. Save the System

At this point, there should be no remaining error messages in the **Messages** tab, and the system is complete. Save the system.

3.7.4.2. Create a Data Pattern Checker Platform Designer System

The data pattern checker system receives a pattern from SDRAM and verifies it against the pattern from the data pattern generator. The pattern reader sends the data to a one-to-two streaming demultiplexer that routes the data to either the custom pattern checker or the PRBS pattern checker. The one-to-two streaming demultiplexer is soft programmable so that the processor can select which pattern checker IP core should verify the data that the pattern reader reads. The custom pattern checker is also soft programmable and is configured to match the same pattern as the custom pattern generator.

Refer to the *Platform Designer Memory Tester* figure for a graphical description.

3.7.4.2.1. Create a New Platform Designer System and Set Up the Clock Source

1. In the Intel Quartus Prime software, click **Tools ➤ Platform Designer** to create a new Platform Designer design.
2. In the **System Contents** tab, Platform Designer shows a clock source instance, **clk_0**. To open the clock source settings, right-click **clk_0**, and then click **Edit**.
3. Turn off **Clock frequency is known** to indicate that, when created, the higher-level hierarchical system that instantiates this subsystem provides the clock frequency.

4. Click **Finish**.
5. Click **File ➤ Save As** to save the Platform Designer system.
6. In the **Save As** dialog box, type **pattern_checker_system**, and then click **Save**.

3.7.4.2.2. Add a Pipeline Bridge

1. In the Library search box, type **bridge** to filter the component list and show only bridge components.
2. Select **Avalon-MM Pipeline Bridge**, and then click **Add**.
3. In the parameter editor, under Parameters, type **11** for the **Address width**. This width accommodates the memory span of all memory-mapped components behind the bridge in this system. As you add the other components in the system, you specify their base addresses within the span of the address space.
4. Accept all other default settings, and then click **Finish**.
   The pipeline bridge is added to your system with the instance name **mm_bridge_0**.
5. On the **System Contents** tab, right-click **mm_bridge_0**, click **Rename**, and then type **mm_bridge**.
6. In the **Clock** column for the **mm_bridge clk** interface, select **clk_0** from the list.
7. To export the **mm_bridge s0** interface, double-click the **Export** column, and then type **slave**.

3.7.4.2.3. Add a Custom Pattern Checker

The custom pattern checker performs the opposite operation of the custom pattern generator. It has a streaming input interface, **st_pattern_input**, that accepts data from the one-to-two streaming demultiplexer. The processor uses the Avalon-MM **csr** slave interface to control the component. The custom packet checker also has a memory-mapped slave interface, **pattern_access**, that the processor uses to program the same patterns as the custom pattern generator component.

1. In the IP Catalog, expand **Memory Test Microcores**, and then double-click **Custom Pattern Checker**.
2. In the parameter editor, accept the default parameters, and then click **Finish**.
3. Rename the instance to **custom_pattern_checker**.
4. Set the **custom_pattern_checker clock** to **clk_0**.
5. Connect the **custom_pattern_checker csr** interface to the **mm_bridge m0** interface.
6. Connect the **custom_pattern_checker pattern_access** interface to the **mm_bridge m0** interface.
7. Assign the **custom_pattern_checker csr** interface to a base address of **0x0420**.
8. Maintain the **custom_pattern_checker pattern_access** interface base address of **0x0000**.

3.7.4.2.4. Add the PRBS Pattern Checker

The PRBS pattern checker performs the opposite operation of the PRBS pattern generator. The processor uses the memory-mapped csr slave interface to control the component. The **st_pattern_input** streaming input accepts data from the one-to-two streaming demultiplexer.

1. In the IP Catalog, expand **Memory Test Microcores**, and then double-click **PRBS Pattern Checker**.
2. In the parameter editor, accept the default parameters, and then click **Finish**.
3. Rename the instance to **prbs_pattern_checker**.
4. Set the **prbs_pattern_checker clock** to **clk_0**.
5. Connect the **prbs_pattern_checker csr** interface to the **mm_bridge m0** interface.
6. Assign the **prbs_pattern_checker csr** interface to a base address of **0x0440**.

3.7.4.2.5. Add a One-to-Two Streaming Demultiplexer

The one-to-two streaming demultiplexer performs the opposite operation of the two-to-one streaming multiplexer. It has a streaming input interface, **st_input**, that accepts data from the pattern reader, and two streaming output interfaces, **st_output_A** and **st_output_B**, that connect to the custom pattern generator and PRBS pattern generator. To allow the processor to program the data route through the component, the system includes the slave interface, **csr**.

1. In the IP Catalog, expand Memory T**est Microcores**, and then double-click **One-to-two Streaming Demux**.
2. In the parameter editor, accept the default parameters, and then click **Finish**.
3. Rename the instance to **one_to_two_st_demux**.
4. Set the o**ne_to_two_st_demux clock** to **clk_0**.
5. Export the **one_to_two_st_demux st_input** interface with the name **st_data_in**.
6. Connect the **one_to_two_st_demux csr** interface to the **mm_bridge m0** interface.
7. Assign the **one_to_two_st_demux csr** interface to a base address of **0x0400**.
8. Connect the **custom_pattern_checker st_pattern_input** interface to the **one_to_two_st_demux st_output_A** interface.
9. Connect the **prbs_pattern_checker st_pattern_input** interface to the **one_to_two_st_demux st_output_B** interface.

3.7.4.2.6. Verify the Memory Address Map

On the **Address Map** tab, verify that the entries in the **Address Map** table match the values in the table below. Red exclamation marks indicate that the address ranges overlap. Correct the base addresses, as appropriate, to ensure there are no overlapping addresses, and your map matches this tutorial's guidelines.

Table 10. Address Map Table

Component	Address
one_to_two_st_demux.csr	0x00000400 - 0x00000407
custom_pattern_checker.csr	0x00000420 - 0x0000042f
custom_pattern_checker.pattern_access	0x00000000 - 0x000003ff
prbs_pattern_checker.csr	0x00000440 - 0x0000045f

3.7.4.2.7. Connect the Reset Signals

You must connect all the reset signals, which eliminates the error messages in the **Messages** tab. Platform Designer allows multiple reset domains, or one reset signal for the system. In the design, you want to connect all the reset signals with the incoming reset signal. To connect all the reset signals, on the System menu, select **Create Global Reset Network**.

At this point in the system design, Platform Designer shows no remaining error messages. If you have any error messages in the **Messages** tab, review the procedures to create this system to ensure you did not miss a step. You can view the reset connections and the timing adapters on the **System Contents** tab, and by selecting **Show System With Platform Designer Interconnect** on the System menu.

3.7.4.2.8. Save the System

At this point, there should be no remaining error messages in the **Messages** tab, and the system is complete. Save the system.

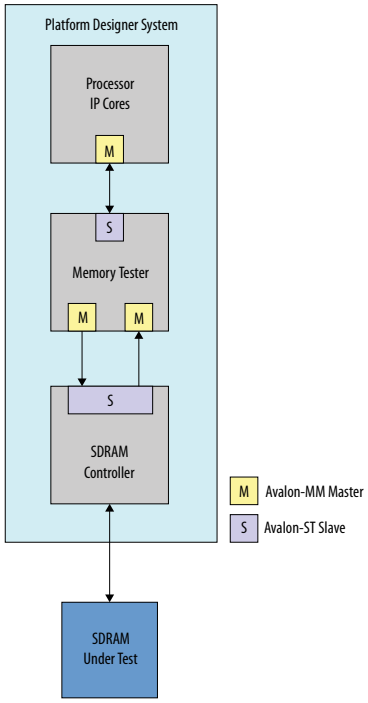
3.7.5. Assemble a Hierarchical System

Hierarchical systems allow you to reuse modular system components. Additionally, hierarchical systems allow you to break large systems into smaller subsystems thus, creating more manageable designs.

The memory tester design includes the following lower-level subsystems:

- **Data pattern generator**—Generates and transmits Avalon-ST data to the memory tester components.
- **Data pattern checker**—Receives and verifies Avalon-ST data from the memory tester components.

Figure 29. Top-Level Memory Tester Design with a Processor and SDRAM Controller



Note: The hierarchical system you create is based on the lower-level **pattern_checker_system.qsys**, and **pattern_generator_system.qsys** subsystems that you created in previous sections. If you did not create these subsystems in the previous section, you can use the completed versions provided with the design files in the **tt_qsys_design\completed_subsystems** directory available from the **Platform Designer Tutorial Design Example** web page. Copy these files to the appropriate `tt_qsys_design\quartus_ii_projects_for_boards\<development_board>` directory for your board.

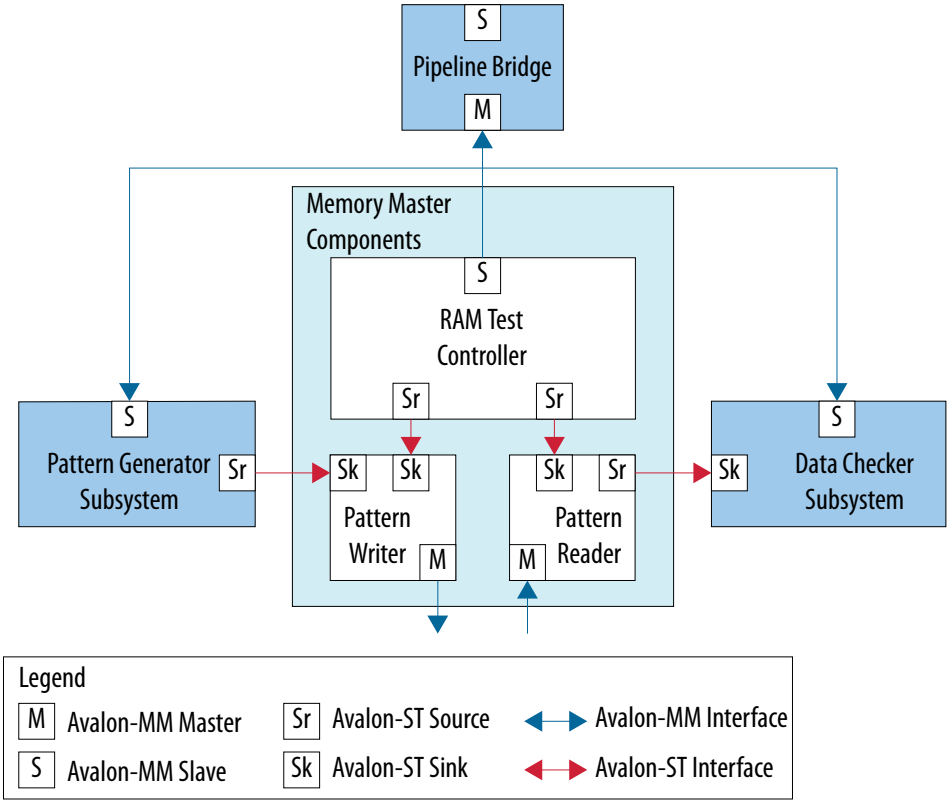
Related Information

Platform Designer Tutorial Design Example

3.7.5.1. Create the Hierarchical Memory Tester System

The memory tester system includes several slave interfaces. However, the memory tester groups the interfaces behind a pipeline bridge that exports a single slave interface to the top-level system. This technique allows the top-level system to access all of the memory-mapped slave ports by reading and writing to a single pipeline bridge slave interface. The bridge also adds a level of pipelining, which can improve timing performance.

Figure 30. Memory Tester Design Interface



1. In Platform Designer, create a new system called, **memory_tester_system**.
2. For the **clk** instance, turn off **Clock frequency is known** to indicate that the higher-level hierarchical system that instantiates this subsystem provides the clock frequency.
3. In the IP Catalog, select the **Avalon-MM Pipeline Bridge** to add to your Platform Designer system.
4. For the **Avalon-MM Pipeline Bridge**, in the parameter editor, type **13** for the **Address width**.
   To accommodate for the address translation from master to slave, that is a byte address as the input, and a word address (4 bytes) as the output, the address width increases from 11.
5. Rename the instance to **mm_bridge**.
6. Set the **mm_bridge_clk** interface to **clk_0**.
7. Export the **mm_bridge s0** interface with the name **slave**.

3.7.5.1.1. Add the Pattern Generator

The custom pattern generator system provides a stream of pattern data via an Avalon-ST source interface. You control the system by accessing the memory locations allocated to each component within the subsystem. The system connects slave ports to a pipeline bridge, which it then exposes outside of the system.

The pattern generator system contains the following components:

- Pipeline bridge
- Custom pattern generator
- PRBS pattern generator
- Two-to-one streaming multiplexer
- Streaming timing adapters

1. In the IP Catalog, under **Project** expand **System**, and then double-click **pattern_generator_system**.
2. In the parameter editor, click **Finish** to accept the default settings.
3. Rename the instance to **pattern_generator_subsystem**.
4. Set the **pattern_generator_subsystem clk** to **clk_0**.
5. Connect the **pattern_generator_subsystem slave** interface to the **mm_bridge m0** interface.
6. Connect the **pattern_generator_subsystem reset** interface to the **clk_0 clk_reset** interface.

3.7.5.1.2. Add the Pattern Checker

The pattern checker system validates data that arrives via an Avalon-ST sink interface. You control the system by accessing the memory locations allocated to each component within the subsystem. The system connects all of the slave ports to a pipeline bridge, which it then exposes outside of the system.

The pattern checker system contains the following components:

- Pipeline bridge
- Custom pattern checker
- PRBS pattern checker
- One-to-two demultiplexer

1. In the IP Catalog, double-click **pattern_checker_system** from the **System** group.
2. In the parameter editor, click **Finish** to accept the default settings.
3. Rename the instance to **pattern_checker_subsystem**.
4. Set the **pattern_checker_subsystem clk** to **clk_0**.
5. Connect the **pattern_checker_subsystem slave** interface to the **mm_bridge m0** interface.
6. Connect the **pattern_checker_subsystem reset** interface to the **clk_0 clk_reset** interface.

3.7.5.1.3. Add Memory Master Components

Memory masters access the SDRAM controller by writing the test pattern to the memory and reading the pattern back for validation. The RAM test controller accepts commands from the processor and controls the memory masters. Each command contains a start address, test length in bytes, and memory block size in bytes. The

RAM test controller segments the commands into smaller block transfers and issues the commands to the read and write masters independently via streaming connections.

When the pattern reader or writer components complete a block transfer, they signal to the RAM test controller that they are ready for another command. The RAM test controller issues the block-sized commands independently, which minimizes the number of idle cycles between memory transfers. The RAM test controller also ensures that the pattern reader never overtakes the pattern writer with respect to the memory locations it is testing, otherwise data corruption occurs.

The SDRAM controller is parameterized to use a local maximum burst length of 2. The pattern reader and writer components are also configured to match this burst length to maximize the memory bandwidth.

Add a Pattern Writer Component

The pattern writer component accepts memory transfer commands from the RAM test controller with the command streaming interface. The **st_data** streaming interface accepts data provided by the design's pattern generator. The mm_data memory-mapped interface writes the pattern data to the SDRAM controller.

1. In the IP Catalog, double-click **Pattern Writer** from the **Memory Test Microcores** group.
2. In the parameter editor, turn on **Burst Enable**.
3. Ensure that the **Maximum Burst Count** is **2**.
4. Ensure that **Enable Burst Re-alignment** is turned on.
5. To accept the other default parameters, click **Finish**.
6. Rename the instance to **pattern_writer**.
7. Set the **pattern_writer clock** to **clk_0**.
8. Connect the **pattern_writer st_data** interface to the **pattern_generator_subsystem st_data_out** interface.
9. Export the **pattern_writer mm_data** interface with the name **write_master**.

Add a Pattern Reader Component

The pattern reader component accepts memory transfer commands from the RAM test controller with the command streaming interface. The mm_data interface reads the pattern data from the SDRAM controller. The st_data interface sends the data read from memory to the design's pattern checker.

1. In the IP Catalog, double-click **Pattern Reader** from the **Memory Test Microcores** group.
2. n the parameter editor, turn on **Burst Enable**.
3. Ensure the **Maximum Burst Count** is **2**.
4. Ensure that **Enable Burst Re-alignment** is turned on.
5. To accept the other default parameters, click **Finish**.
6. Rename the instance to **pattern_reader**.

7. Set the **pattern_reader clock** to **clk_0**.
8. Connect the **pattern_reader st_data** interface to the **pattern_checker_subsystem st_data_in** interface.
9. Export the **pattern_reader mm_data** interface with the name **read_master**.

Add a RAM Test Controller

The RAM test controller contains two streaming command interfaces; write_command and read_command, that send commands to the pattern reader and pattern writer components. These streaming interfaces issue commands effectively because Avalon-ST interfaces offer low latency and a simple handshaking protocol, as well as because the processor accesses a slave port, csr, to write commands to the controller.

1. In the IP Catalog, double-click **RAM Test Controller** from the **Memory Test Microcores** group.
2. In the parameter editor, click **Finish** to accept the default parameters.
3. Rename the instance to **ram_test_controller**.
4. Set the **ram_test_controller** clock to **clk_0**.
5. Connect the **ram_test_controller write_command** interface to the **pattern_writer_command** interface.
6. Connect the **ram_test_controller read_command** interface to the **pattern_reader_command** interface.
7. Connect the **ram_test_controller csr** interface to the **mm_bridge m0** interface.

Do not use the Generation tab at this point in the tutorial to generate HDL code for these subsystems. You must generate files for the entire top-level system, which includes all the subsystems. The batch script provided for you to program the device requires that only one system is generated in the project directory. The top-level design includes a Nios II subsystem, and the Nios II software build tools require the SOPC Information File (.sopcinfo) to be generated for the top-level design. If there are multiple .sopcinfo files, the batch script to program the device fails with an error from the software build tools.

3.7.5.1.4. Connect the Reset Signals

You must connect all the reset signals, which eliminates the error messages in the **Messages** tab. Platform Designer allows multiple reset domains, or one reset signal for the system. In the design, you want to connect all the reset signals with the incoming reset signal. To connect all the reset signals, on the System menu, select **Create Global Reset Network**.

At this point in the system design, Platform Designer shows no remaining error messages. If you have any error messages in the **Messages** tab, review the procedures to create this system to ensure you did not miss a step. You can view the reset connections and the timing adapters on the **System Contents** tab, and by selecting **Show System With Platform Designer Interconnect** on the System menu.

3.7.5.1.5. Verify the Memory Address Map

To ensure that the memory map of the system you create matches the memory map of other components, you must verify the base addresses for the memory tester system. In Platform Designer, on the Address Map tab, verify that the entries in

Address Map table match the values in Table 3–1. Red exclamation marks indicate that the address ranges overlap. Correct the base addresses, as appropriate, to ensure there are no overlapping addresses.

Table 11. Address Map Table

Component	Base Address	Address
mm_bridge_0.s0	N/A	N/A
pattern_generator_subsystem.slave	0x0	0x00000000 – 0x000007ff
pattern_checker_subsystem.slave	0x1000	0x0001000 – 0x000017ff
ram_test_controller.csr	0x800	0x00000800 – 0x0000081f

3.7.5.1.6. Save the System

At this point, there should be no remaining error messages in the **Messages** tab, and the system is complete. Save the system.

3.7.5.2. Complete the Top-Level System

1. In Platform Designer, open the **top_system.qsys** file from the `tt_qsys_design\quartus_ii_projects_for_boards\<development_board>` directory.
   The top-level system is set up for your development board, with an external clock source, a processor system, and an SDRAM controller. You can view the clocks in top-level system on the **Clock Settings** tab, and the partially-completed system connections on the **System Contents** tab.
2. In the IP Catalog, double-click **memory_tester_system** from the **System** group.
3. Click **Finish** to accept the default parameters, and to add the memory tester system to the top-level system.
4. Rename the system to **memory_tester_subsystem**.
5. On the **System Contents** tab, use the arrows to move the **memory_tester_subsystem** up between the **cpu_subsystem** and the **sdram**. Since the **cpu_subsystem** controls the **memory_tester_subsystem**, and the **memory_tester_subsystem** controls the **sdram**, this positioning allows you to more easily visualize system performance.
6. Set the **memory_tester_subsystem** clk to either the **sdram_sysclk** (for ALTMEMPHY-based designs), or **sdram_afi_clk** (for UniPHY-based designs). Some boards have an FPGA and SDRAM device that use either the Intel DDR or DDR2 SDRAM Controller with ALTMEMPHY; others use the Intel DDR3 SDRAM controller with UniPHY.
7. Connect the **memory_tester_subsystem reset** interface to the **ext_clk clk_reset** interface.
8. Connect the **memory_tester_subsystem reset** interface to the **cpu_subsystem cpu_jtag_debug_reset** interface.
   This design exports the Nios II processor JTAG debug reset output interface, **jtag_debug_module_reset**, from the **cpu_subsystem** with the interface name **cpu_ jtag_debug_reset**. The design must connect this Nios II reset output to any component reset inputs that require resetting by the Nios II processor code or JTAG interface, and also to the Nios II processor's reset input interface. The

c**pu_subsystem cpu_reset** interface connects to the Nios II processor's reset input interface. The **top_level.qsys** file connects the **cpu_jtag_debug_reset** interface to the **cpu_reset** interface.

9. Connect the **memory_tester_subsystem write_master** and **read_master** interfaces to either the **sdram s1** interface (for ALTMEMPHY-based designs), or **sdram avl** interface (for UniPHY-based designs).
10. Connect the **memory_tester_subsystem slave** interface to the **cpu_subsystem master** interface.
11. Maintain the base addresses of **0x0** for the **memory_tester_subsystem slave interface**, and for either the **sdram s1** interface (for ALTMEMPHY-based designs), or **sdram avl** interface (for UniPHY-based designs).

The two slave interfaces can use the same address map range because different masters control them. The **cpu_subsystem master** interface controls the **memory_tester_subsystem**, and the **memory_tester_subsystem write_master** and **read_master** interfaces control the **sdram** interface.

3.7.6. Viewing the Memory Tester System in Platform Designer

You can use the **Hierarchy** tab, accessed from the View menu, to show the complete hierarchy of your design. The **Hierarchy** tab is a full system hierarchical navigator, which expands the system contents to show modules, interfaces, signals, contents of subsystems, and connections. The graphical interface of the **Hierarchy** tab displays a unique icon for each element represented in the system, including interfaces, directional pins, IP blocks, and system icons that show exported interfaces and the instances of components that make up a system.

Click **Generate ➤ HDL Example** to view the HDL for an example instantiation of the system. The HDL example lists the signals from the exported interfaces in the system. The signal names are the exported interface name followed by an underscore, and then the signal name specified in the component or IP core. Most of the signals connect to the external SDRAM device.

3.7.7. Compiling and Downloading Software to a Development Board

Intel recommends that you download the memory tester system to a development board to complete the design process and test the memory interface of the board. If you do not have a development board you can follow the steps provided in the accompanying **readme.txt file** to learn more details about porting designs to FPGA devices or boards.

The Intel provided software tests the memory using various test parameters and patterns, and is scripted for compilation and download to the board.

1. To download the top-level system to a development board, in Platform Designer, click **Generate ➤ Generate**.
2. Select the language for **Create HDL design files for synthesis**, and turn off the option to create a Block Symbol File (**.bsf**).
3. Click **Generate**. Platform Designer generates HDL files for the system and the Intel Quartus Prime IP File (**.qip**) that provides the list of required HDL files for the Intel Quartus Prime compilation.
4. When Platform Designer completes the generation, click **Close**.

5. In the Intel Quartus Prime software, on the Project menu, click **Add/Remove Files in Project** and verify that the newly-generated **.qip** file, **top_system.qip**, and the timing constraints file, **my_constraints.sdc** appear in the **Files** list.
6. Click **Processing ➤ Start Compilation**. When compilation completes, click **OK**.
7. Connect the development board to a supported programming cable.
8. Click **Tools ➤ Nios II Command Shell [gcc4]**.
9. Type the following command to emulate your local c:/ drive for your Windows environment: **cd /cygdrive/c/**.
10. Navigate to the `quartus_ii_projects_for_boards\<development_board>\software` directory.
11. Type the following command at the Nios II command Shell: **./batch_script.sh**. The batch script compiles the Nios II software and downloads the SRAM Object File (**.sof**) programming file to the FPGA.

The terminal window shows messages indicating the progress. If you see error messages related to the JTAG chain, check your programming cable installation and board setup to ensure that it is set up correctly.

After the script configures the FPGA, it downloads the compiled Nios II software to the board and establishes a terminal connection with the board. The test software performs test sweeps on the SDRAM by varying the following parameters:

- Pattern type
- Memory block size
- Memory block trail distance (number of blocks by which the pattern reader trails the pattern writer)
- Memory span tested

Ensure that you have only one set of generated system files in the project directory, otherwise the batch script to program the device fails with an error from the software build tools.

The memory throughput values appear in the command terminal as the memory is tested. These values are reported in hexadecimal and represent the number of clock cycles required to test the entire SDRAM address span. The output is restricted to hexadecimal due to a small software library that prints the characters to the terminal. Because the memory tester system writes to the memory and then reads it back, the number of bytes it accesses and reports in the transcript window is double the memory span. This number varies depending on the span of memory being tested for your memory device. Knowing the data width of the memory interface, the number of bytes transferred, and the number of clock cycles for the transfer, you can determine the memory access efficiency.

The SDRAM controller in the top-level Platform Designer system has a 32-bit local interface width, therefore **memory data width in bytes** is 4 bytes for the tutorial design.

```
Efficiency = 100 × total bytes transferred/(memory data width in bytes × total clock cycles)
```

The memory test runs until the design finishes testing the complete memory. To end the test early, type Ctrl+C in the command window. To calculate the efficiency for the last throughput numbers in, convert the hexadecimal numbers to decimal, as follows:

- 0x4000000 bytes transferred is 0d67108864 total bytes transferred
- 0x107d856 clock cycles is 0d17291350 total clock cycles

Therefore, the efficiency for this example is:

```
100 × 67108864 / (4 × 17291350) = 97.0%
```

3.7.8. Debugging Your Design

If the memory test starts but does not complete successfully, the terminal displays failure messages. If you see failure messages from the memory test, review the previous sections and check that you have completed all of the instructions in this tutorial successfully. A missed connection or incorrect memory address assignment may cause the tester design to fail on the board.

Intel provides completed systems, so that you can verify your system designs. You can copy the completed systems into the project directory with different names, so that you can open two different instances of Platform Designer for a side-by-side comparison. Alternatively, you can replace your systems with the provided completed systems to run the memory tester design successfully.

3.7.9. Verifying Hardware in System Console

You can use the Intel Quartus Prime System Console to verify your system design. The design example files include scripts that exercise your system using System Console Tcl commands. The example uses a JTAG-to-Avalon Master Bridge component to drive the slave components, instead of a Nios II processor system.

The `\quartus_ii_projects_for_boards\<development_board>\system_console` directory contains the **run_sweep.tcl**, **base_address.tcl**, and **test_cases.tcl** scripts. You use these scripts to set up and run memory tests on the development board projects. You can view the scripts to help you understand the System Console commands that drive the slave component registers. The scripts work with any board, if you keep the same Platform Designer system structure.

The `run_sweep.tcl` file is the main script, which calls the other two scripts. The `base_address.tcl` file includes information about the base addresses of the slave components from the previous chapters. If you change the base addresses of the slave components, you must also change the addresses in the `base_address.tcl` file. The `test_cases.tcl` file includes settings for memory span, memory block sizes, and memory block trail distance.

The **run_sweep.tcl** file contains Tcl commands for the following actions:

- Initialize the components
- Adjust test parameters
- Start the PRBS pattern checker, PRBS pattern generator, and RAM controller
- Continuously poll the stop and fail bits in the PRBS checker

Related Information

Analyzing and Debugging Designs with System Console

3.7.9.1. Open the Tutorial Project

You can use completed design files in the `tt_qsys_design \quartus_ii_projects_for_boards\<development_board>` directory.

1. Open the Intel Quartus Prime project in the project directory for your development board type.
2. In Platform Designer, open **top_system.qsys** in the project directory for your development board type.

3.7.9.2. Add the JTAG-to-Avalon Master Bridge

The JTAG-to-Avalon master bridge acts as a bridge between the JTAG interface and the system's memory tester.

1. In the IP Catalog select **JTAG to Avalon Master Bridge**, and then click **Add**.
2. In the parameter editor, click **Finish** to accept the default parameters.
3. Rename the instance to **jtag_to_avalon_bridge**.
4. Connect the **jtag_to_avalon_bridge master** interface to the **memory_tester_subsystem slave** interface.
5. Set the **jtag_to_avalon_bridge clk** domain to **sdram_sysclk**.
6. Connect the **jtag_avalon_bridge clk_reset** interface to the **ext_clk clk_reset** interface.
7. Connect the **jtag_avalon_bridge clk_reset** interface to either the **sdram reset_request_n** interface (for ALTMEMPHY-based designs), or **sdram afi_reset** interface (for UniPHY-based designs).
8. Connect the **jtag_avalon_bridge master_reset** interface to the **memory_tester_subsystem reset** interface, and to either the **sdram soft_reset_n** interface (for ALTMEMPHY-based designs), or **sdram soft_reset** interface (for UniPHY-based designs).
9. To disable the **cpu_subsystem** system, in the **Use** column, turn off **Use**, since you are replacing its function with the bridge and System Console.
10. Save the **jtag_to_avalon_bridge** system.

3.7.9.3. Debug with System Console

The design example scripts test the memory in loops for different block sizes, that is, the number of bytes to group together in a single instance of back-to-back reads or writes. The scripts also test the memory in loops for different memory block trails, that is, the number of blocks by which the pattern reader trails the pattern writer.

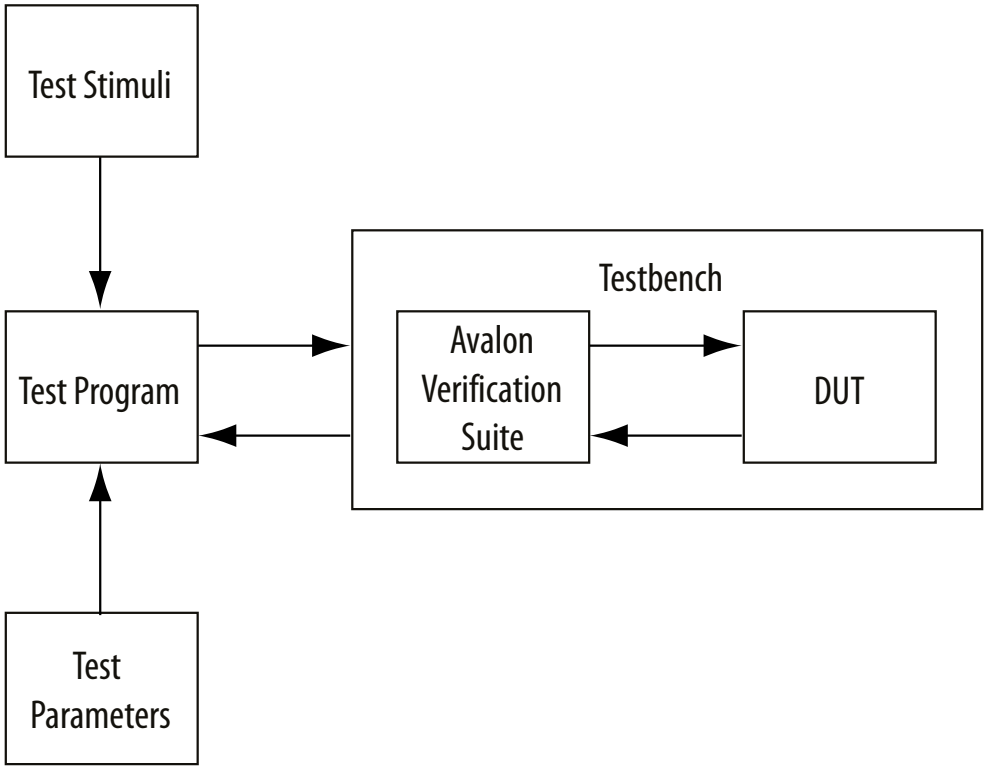
1. To download the programming file to your development board, in Platform Designer, click **Generate ➤ Generate**.
2. Select the language for **Create HDL design files for synthesis**.
3. Click Generate. Platform Designer generates HDL files for the system and the **.qip** file, which provides the list of required HDL files for the Intel Quartus Prime compilation.

4. When Platform Designer completes the generation, click **Close**.
5. In the Intel Quartus Prime software, click **Project ➤ Add/Remove Files in Project**, and verify that the project contains the **top_system.qip**.
6. Click **Processing ➤ Start Compilation**. When compilation completes, click **OK**.
7. Connect the development board to a supported programming cable.
8. Click **Tools ➤ Programmer**.
9. Check that the Programmer displays the correct programming hardware. Otherwise, click **Hardware Setup** and select the correct programming hardware, and then click **Close**.
10. To program the device, click **Start**.
11. In Platform Designer, click **Tools ➤ System Console**.
12. Before you execute scripts in System Console, navigate to the directory for the Tcl scripts, and then in Platform Designer System Console window, click **File ➤ Execute Script**.
13. To start the memory tests, run the **run_sweep.tcl** file from the `tt_qsys_design\quartus_ii_projects_for_boards\<development_board> \system_console` directory.
    When you run the **run_sweep.tcl** script, the System Console displays the progress of the tests in the **Messages** tab. The tests perform test sweeps on the SDRAM by varying the memory block size and memory block trail distance. When the tests finish successfully, Platform Designer generates a message that reports successful completion.

3.7.10. Simulating Custom Components

You can simulate a custom component with Platform Designer and the Avalon Verification IP Suite. You use Platform Designer to generate a testbench system for the design under test, and then perform a functional simulation with the ModelSim simulator. The Platform Designer-generated testbench uses the Avalon Verification IP Suite components.

Figure 31. Typical Platform Designer Test Environment



3.7.10.1. Generate a Testbench System in Platform Designer

The custom pattern generator generates high-speed streaming data for testing memory devices. The soft-programmable custom pattern generator can generate multiple test patterns, and is programmed with the pattern data and pattern length. When the end of the pattern is reached, the custom pattern generator cycles back to the first element of the pattern.

If you do not want to use the Platform Designer-generated testbench system, you can create your own Platform Designer testbench system by adding the Avalon Verification Suite Bus Functional Models (BFMs) or your own models for simulation. You can also generate a Platform Designer simulation model for the design or Platform Designer system under test, and use your own custom HDL testbench to provide the simulation stimulus.

3.7.10.1.1. Create a New Platform Designer System for the Design Under Test

1. In the Intel Quartus Prime software, open the Intel Quartus Prime Project File, **qsys_sim_tutorial.qpf**, from the **\simulation_tutorial** directory.
2. In Platform Designer, click **File ➤ New System** to create a new Platform Designer design.
3. To remove the clock source, which is not needed for this design, right-click **clk_0**, and then click **Remove**.

4. In the IP Catalog, select **Custom Pattern Generator** from the **Memory Test Microcores** group, and then click **Add**.
5. In the parameter editor, click **Finish** to accept the default parameters.
6. Rename the instance to **pg** to provide a short instance name for the pattern generator.

3.7.10.1.2. Export Design Under Test

1. In Platform Designer, on the **System Contents** tab, in the **Export** column, for each interface click **Double-click to export**, and maintain the default export names.
2. Save the system as **pattern_generator**.

3.7.10.1.3. Generate a Testbench System

1. In Platform Designer click **Generate ➤ Generate Testbench System**.
2. Under **Testbench System**, for **Create testbench Platform Designer system**, select **Standard, BFMs for standard Platform Designer interfaces**.
3. Under **Synthesis**, select None for **Create HDL design files for synthesis**, and turn off **Create block symbol file (.bsf)**.
4. Click **Generate**.
5. After Platform Designer generates the testbench, click **Close**.

Platform Designer generates this testbench system in the **\simulation_tutorial \pattern_generator\testbench** directory.

You can generate the simulation model for the Platform Designer testbench system at the same time by turning on Create testbench simulation model. However, the Platform Designer-generated testbench system's components names are assigned automatically and you may want to control the instance names to make it easier to run the test program for the BFMs. In this tutorial, you edit the Platform Designer testbench system before generating the simulation model.

3.7.10.1.4. Generate Testbench System's Simulation Models

In this section, you open the generated Platform Designer testbench system and rename the BFM component instance names to ensure the testbench names match the test program provided with the tutorial design files. Additionally, you generate the testbench's simulation model.

1. In Platform Designer, open the testbench system, **pattern_generator_tb.qsys**, from the `simulation_tutorial\pattern_generator\testbench` directory.
2. On the System Contents tab, rename the instance as they appear in Table 5–1.

Platform Designer-Generated Components' Names	New Instance Name
pattern_generator_inst	DUT
pattern_generator_inst_pg_clock_bfm	clock_source
pattern_generator_inst_pg_reset_bfm	reset_source
	continued...

Platform Designer-Generated Components' Names	New Instance Name
pattern_generator_inst_pg_csr_bfm	csr_master
pattern_generator_inst_pg_pattern_access_bfm	pattern_master
pattern_generator_inst_pg_pattern_output_bfm	pattern_sink

3. Double-click a BFM component to open the parameter editor and view its settings. These BFM components are available in the Avalon Verification Suite group in the library. If necessary, you can change the parameters for the BFMs to ensure adequate test coverage for your design.
   The Platform Designer-generated testbench matches inserted BFMs with the exported interfaces from the design that they drive. The test program that provides stimulus to the BFMs must account for the matching interface. For example, an exported Avalon-MM slave interface (which expects word-aligned addresses) is connected to an Avalon master BFM, which expects and transacts word-aligned addresses instead of the byte or symbol addresses that are default for Avalon masters.
4. Click **Cancel** to close the parameter editor without making changes.
5. In the **Generation** dialog box, under **Simulation**, for **Create simulation model**, select **Verilog**.
6. Under **Testbench System**, select **None** for **Create testbench Platform Designer system** and **Create testbench simulation model**.
7. Under **Synthesis**, select **None** for **Create HDL design files for synthesis**, and turn off **Create Block design files (.bsf)**.
8. Save the system.
9. Click **Generate**.
10. After Platform Designer generates the testbench, click **Close**.

Platform Designer generates the testbench system's simulation models in the **\simulation_tutorial\pattern_generator\testbench\pattern_generator_tb\simulation** directory.

Platform Designer generates the simulation models and a ModelSim simulation script (**msim_setup.tcl**), which compiles the required files for simulation and sets up commands to load the simulation in the ModelSim simulator. You can run this ModelSim script in ModelSim to compile, elaborate, or load for simulation.

In this tutorial, there is an external test program to provide simulation stimulus. The tutorial design files include a simulation script, **load_sim.tcl** that compiles the top-level simulation file and test program, and calls the Platform Designer-generated script to compile the required files.

3.7.10.2. Run Simulation In the ModelSim Software

You can run a simulation in the ModelSim software on the testbench that you created. To complete this simulation you use the test program provided in the design files. The test begins by writing a walking ones pattern to the design under test.

This test program performs the following actions:

- Reads a pattern file.
- Writes the pattern to the design under test via the pattern master BFM.
- Sets various design under test options via the CSR master BFM.
- Starts the design under test pattern generation.
- Collects data generated by the design under test.
- Compares the results against the original pattern file.

3.7.10.2.1. Set Up the Simulation Environment

This tutorial includes test program files that you can use with the Platform Designer-generated testbench and ModelSim simulation script. To learn more about Platform Designer simulation support, open and review the simulation script, **\simulation_tutorial\load_sim.tcl**. After your review of the script, close the script without making changes.

The **load_sim.tcl** script sets simulation variables to set up the correct hierarchical paths in the Platform Designer-generated simulation model and ModelSim script. Additionally, the script identifies the top-level instance name for the simulation and provides the path to the location of the Platform Designer-generated files. Some functions, such as memory initialization, rely on correct hierarchical paths names in the simulation model.

The **load_sim.tcl** script performs the following actions:

- Sources the Platform Designer-generated ModelSim simulation script, **msim_setup.tcl**.
- Uses the command aliases defined in the **msim_setup.tcl** script to compile and elaborate the files for the Platform Designer testbench simulation model.
- Compiles and elaborates the extra simulation files for the tutorial—the test program and top-level simulation file that instantiates the test program.
- Loads the **wave.do** file that provides signals for the ModelSim waveform view.

3.7.10.2.2. Run the Simulation

1. Start the ModelSim software.
2. Click **File ➤ Change Directory**, browse to the **\simulation_tutorial** directory, and then click **OK**.
3. Click **Compile ➤ Compile Options**.
4. Click the **Verilog & SystemVerilog** tab, select **Use SystemVerilog**, and then click **OK**.
5. Click **File ➤ Load**
   Ensure you activate the ModelSim Transcript window, otherwise the Load function is disabled.
6. Select the **load_sim.tcl** script, and then click **Open**.
   The warning messages relate to unused connections in an ALTSYNCRAM megafunction. Because these ports are not used, you can ignore the warning messages.

7. Run the simulation for 40us. To run the simulation, in the ModelSim Transcript window type the following command: **run 40us**.
   You can run the **h** command to show the available options for the **msim_setup.tcl script**.
8. Observe the results.

```
INFO: top.tb.reset_source.reset_deassert: Reset deasserted
INFO: top.pgm: Starting test walking_ones.hex
INFO: top.pgm.read_file: Read file walking_ones.hex success
INFO: top.pgm.read_file: Read file walking_ones_rev.hex success
INFO: top.pgm: Test walking_ones.hex passed
```

9. To run the low frequency test, modify **\simulation_tutorial\test_include.svh** according to Table 12 on page 116.

Table 12. Values for Low Frequency Pattern Test

Macro	New Value
PATTERN_POSITION	0
NUM_OF_PATTERN	2
NUM_OF_PAYLOAD_BYTES	256
FILENAME	low_freq.hex
FILENAME_REV	low_freq_rev.hex

10. Reload the **load_sim.tcl** script, run the simulation for 40us, and observe the result in the Transcript window.

```
INFO: top.pgm: Starting test low_freq.hex
INFO: top.pgm.read_file: Read file low_freq.hex success
INFO: top.pgm.read_file: Read file walking_ones_rev.hex success
INFO: top.pgm: Test low_freq.hex passed
```

11. To run the random number pattern test, modify **\simulation_tutorial \test_include.svh** according to Table 13 on page 116.

Table 13. Values for Random Number Pattern Test

Macro	New Value
PATTERN_POSITION	32
NUM_OF_PATTERN	64
NUM_OF_PAYLOAD_BYTES	1024
FILENAME	random_num.hex
FILENAME_REV	random_num_rev.hex

12. Reload the **load_sim.tcl** script, and run the simulation for 40us to observe the following results.

```
INFO: top.pgm: Starting test random_num.hex
INFO: top.pgm.read_file: Read file random_num.hex success
INFO: top.pgm.read_file: Read file random_num_rev.hex success
INFO: top.pgm: Test random_num.hex passed
```

3.7.11. View a Diagram of the Completed System

You set up the simulation environment for the custom pattern generator component and used BFM test code to perform simulation. You can test your own custom Platform Designer components with this method to verify their functionality before you integrate them into a complete system. You can also create a testbench system for a complete Platform Designer system with this method, and test your top-level system behavior with BFMs.

Related Information

Platform Designer Tutorial Design Example

3.8. Hardware System Design with Intel Quartus Prime and Platform Designer Revision History

Document Version	Changes
2017.06.12	Added new sections: • *Hardware Development Tutorial* • *System Design Tutorial*
2016.12.19	Initial release.

Related Information

Document Revision History for Embedded Design Handbook on page 6

4. Software System Design with a Nios II Processor

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 Intel 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.

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

4.1. Nios II Command-Line Tools

The Intel 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 Intel-provided command-line tools for developing Nios II programs in "Intel-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 Intel 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

4.1.1. Intel Command-Line Tools for Board Bringup and Diagnostics

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

4.1.1.1. 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.

4.1.1.1.1. 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. jtagconfig Example Response

```
$ jtagconfig -n
1) Intel FPGA Download Cable [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 14. Interpretation of jtagconfig Command Response

Value	Description
`Intel FPGA Download Cable [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.
`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 Intel FPGA passes through this hub, including advanced debugging capabilities such as the Signal Tap 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.

4.1.1.2. 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 (Intel FPGA Download 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.

4.1.1.2.1. 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 Intel 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.

4.1.1.3. 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 *Intel Quartus Prime Handbook*, on-line training is available.

Related Information

Analyzing and Debugging Designs with System Console

4.1.2. Intel 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

4.1.2.1. 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 Intel Quartus Prime Programmer flow for EPCS.

4.1.2.1.1. 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 125, 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 Platform Designer. 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 123.
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
   ```

4.1.2.2. 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 5. 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.

4.1.2.2.1. 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
```

4.1.3. Intel Command-Line Tools for Software Development and Debug

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

4.1.3.1. 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 121 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"`).

4.1.3.2. 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.

4.1.3.2.1. 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.

4.1.3.3. 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 6. 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.
```

4.1.3.3.1. 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
```

4.1.3.4. 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.

4.1.3.4.1. 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.

4.1.3.5. 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
```

4.1.3.5.1. 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 "Intel FPGA Download Cable [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 7. 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.

4.1.4. Intel 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 137

4.1.4.1. 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.

4.1.4.2. 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.

4.1.5. 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 > Intel FPGA > 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 8. 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.

4.1.6. 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 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.

4.1.6.1. 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 136 and the `nios2-elf-nm` command described in nios2-elf-nm on page 135.

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 9. 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
```

4.1.6.1.1. 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.

4.1.6.2. 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 127.

4.1.6.3. 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.

4.1.6.3.1. 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 10. 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 15. 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.

4.1.6.4. 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.

4.1.6.4.1. 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 11. nios2-elf-ar Command Response

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

4.1.6.5. Linker

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

4.1.6.5.1. 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: Intel 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.

4.1.6.6. nios2-elf-size

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

4.1.6.6.1. 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 12. 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
```

4.1.6.7. nios2-elf-strings

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

4.1.6.7.1. nios2-elf-strings Usage Example

The command has a single required argument:

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

4.1.6.8. 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**).

4.1.6.8.1. nios2-elf-strip Usage Example

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

4.1.6.8.2. 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.

4.1.6.9. 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 130.

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.

4.1.6.10. 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

4.1.6.11. nios2-elf-gcc and g++

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

4.1.6.11.1. 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>
```

4.1.6.11.2. 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 13. 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
```

4.1.6.12. 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.

4.1.6.12.1. 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
```

4.1.6.12.2. 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.

4.1.6.13. nios2-elf-nm

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

4.1.6.13.1. 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`

4.1.6.13.2. 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.

4.1.6.14. 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 16. -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
binary	Raw binary format Useful for creating binary images for storage in flash on your embedded system
ihex	Intel 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.

4.1.6.14.1. 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
```

4.1.6.15. 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.

4.1.6.15.1. 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
```

4.1.6.16. 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 131 or type `nios2-elf-ar --help` in a Nios II command shell.

4.2. Developing Nios II Software

This section provides in-depth information about software development for the Intel 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 Intel website.
- Documentation located on the Documentation: Nios II Processor page of the Intel 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 Intel'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 Intel Quartus Prime software. For example, if the Intel Quartus Prime software is installed on the Windows operating system, and the root directory of the Intel 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

4.2.1. 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 Intel-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

- Intel System on a Programmable Chip (Platform Designer) Solutions on page 20
- Nios II Software Development Process on page 140
- rocketboards.org

4.2.1.1. 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.

4.2.1.1.1. 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 Intel-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 Intel HAL, refer to The Hardware Abstraction Layer section of the *Nios II Gen2 Software Developer's Handbook*.
- Nios II SBT

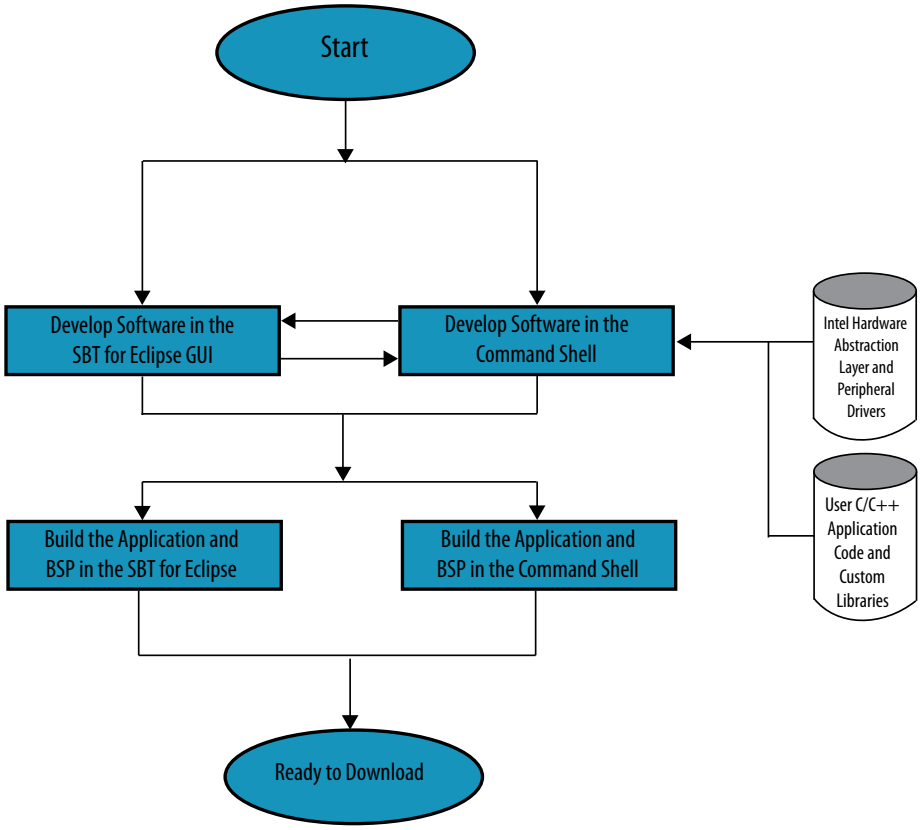
  The Nios II SBT development flow is a scriptable development flow. It includes the following user interfaces:

  — 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 32. Nios II Software Development Flows: Developing Software



Intel 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: Intel recommends that you use an Intel 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 120
- Overview of the Hardware Abstraction Layer
- Developing Nios II Software on page 137

4.2.1.1.2. 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. Intel 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

4.2.1.2. 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
   - Platform Designer
   - Intel 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 Intel Quartus Prime and Platform Designer 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 Platform Designer 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 Intel-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 Intel-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*.

4.2.2. 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.

4.2.2.1. 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.

Intel 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 Intel'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.

4.2.2.2. 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, Intel 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 Intel 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. Intel recommends that you use a version control system to maintain distinct versions of your source files as you develop your project.

Related Information

- Design Examples in the Intel Design Store
- Nios II Gen2 Hardware Development Tutorial

4.2.2.3. 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 "Intel-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

4.2.2.3.1. 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

4.2.2.3.2. 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

4.2.2.4. 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

4.2.2.4.1. 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 Intel 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 > 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 Intel 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 Intel 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 **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

4.2.2.4.2. 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.

4.2.2.4.3. 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.

Intel 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

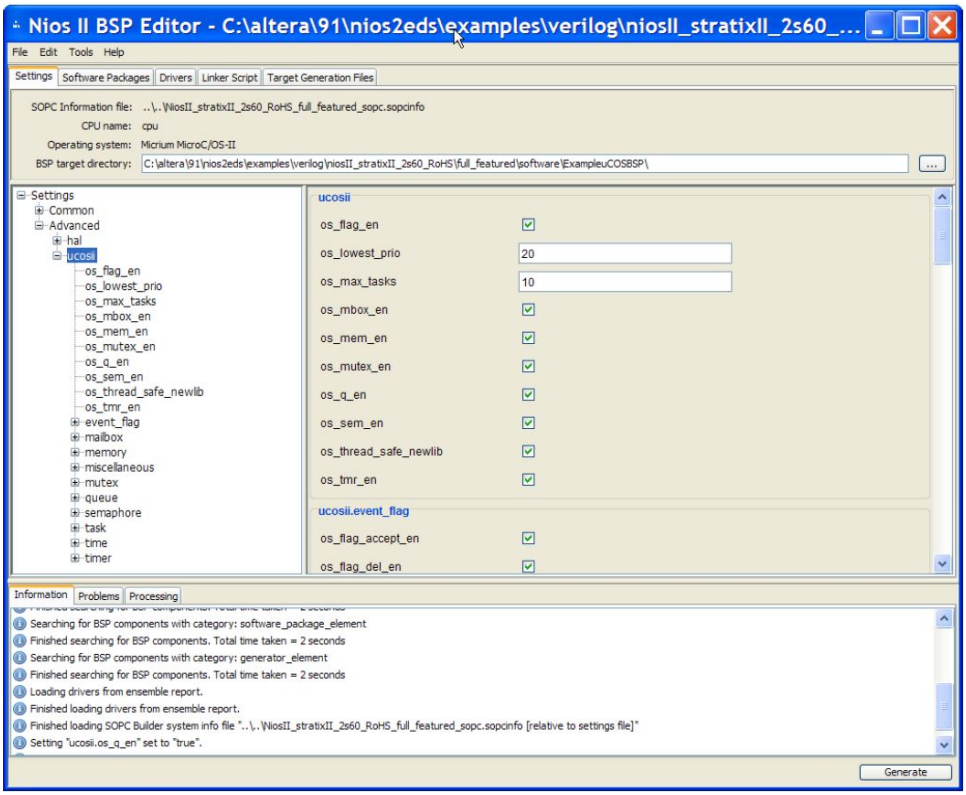
If you use the MicroC/OS-II RTOS environment, be aware of the following properties of this environment:

- **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".

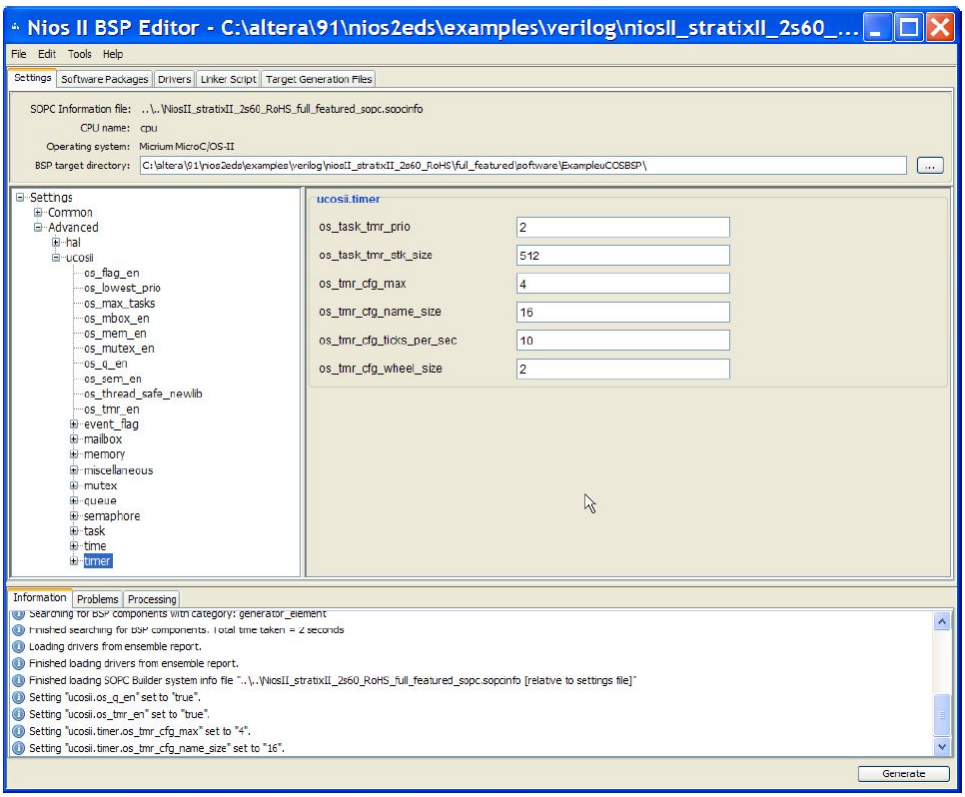
You can configure MicroC/OS-II with the BSP Editor. Figure 2–1 shows how you enable the MicroC/OS-II timer and queue code. The figure below shows how you specify a maximum of four timers for use with MicroC/OS-II.

Figure 33. 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 34 on page 149: it enables the timer and queue code, and specifies a maximum of four timers.

Figure 34. Configuring MicroC/OS-II for Four Timers in BSP Editor



Example 14. 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 163

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.

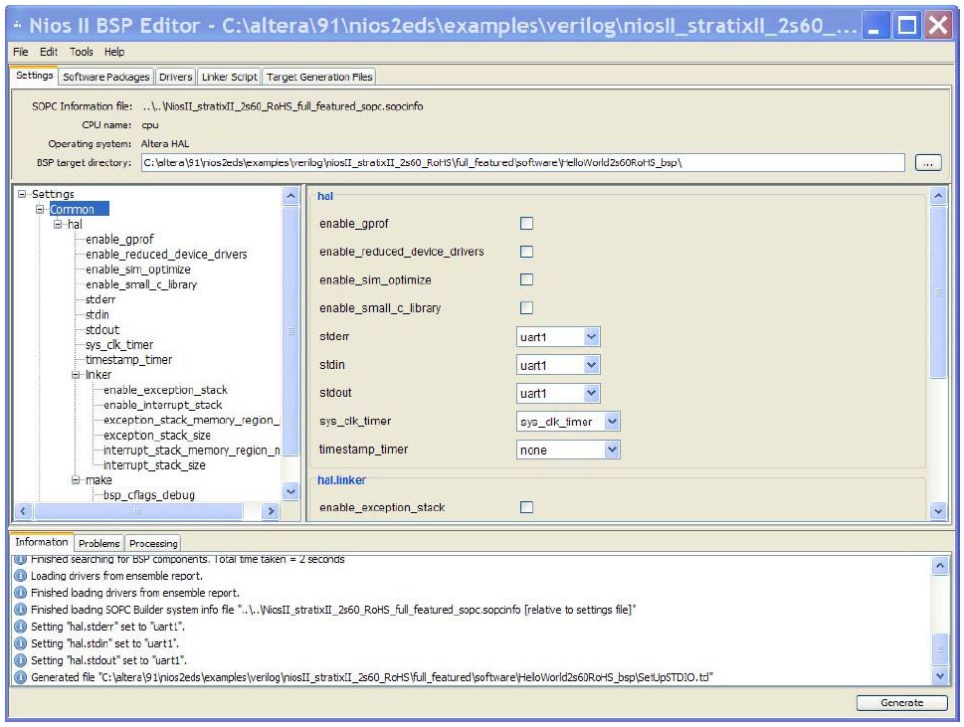
  Note: For more information about BSP settings, refer to "Settings Managed by the Software Build Tools" in the Nios II Software Build Tools Reference chapter of the *Nios II Gen2 Software Developer's Handbook*.

- **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 35. Configuring HAL stdio Device in BSP Editor



Example 15. 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 163

Adding Software Packages

Intel 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 Intel 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

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 non-default 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 16. Tcl Script Exported by BSP Editor

```
#
# This is a generated Tcl script exported
# by a user of the Intel FPGA Nios II BSP Editor.
#
# It can be used with the Intel FPGA '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:

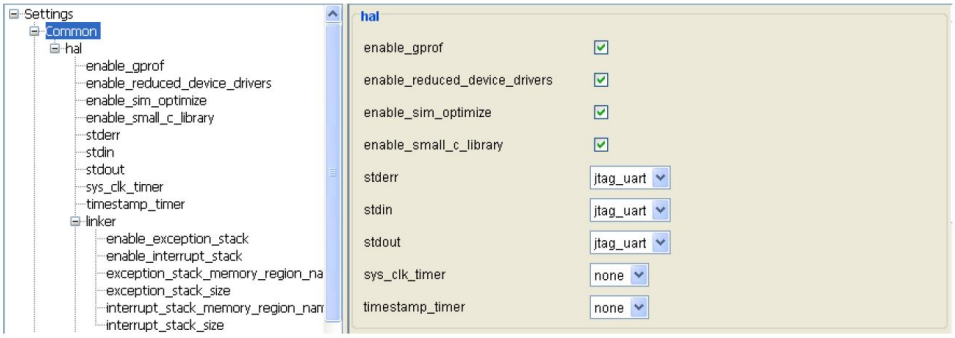
1. With any text editor, create a new file called **example.tcl**.
2. Insert the contents of the example below in the file.
3. In the Nios II BSP Editor, in the File menu, click **New BSP**.

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 **Intel FPGA HAL**.
6. In the **Additional Tcl script** box, navigate to **example.tcl**.
7. In the **Platform Designer 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 17. 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 36. Nios II BSP Settings Configured with example.tcl



Do not attempt to import an Intel FPGA 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 152

4.2.2.4.4. 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

Design Examples in the Intel Design Store

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.

4.2.2.4.5. 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

4.2.2.4.6. 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 395

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 395

4.2.2.5. Ensuring Software Project Coherency

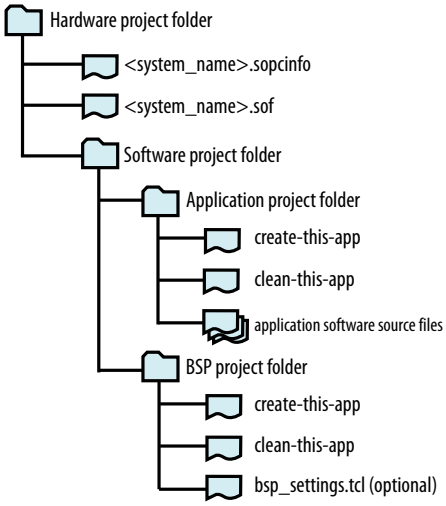
In some engineering environments, maintaining coherency between the software and system hardware projects is difficult. For example, in a mixed team environment in which a hardware engineering team creates new versions of the hardware, independent of the software engineering team, the potential for using 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.

4.2.2.5.1. 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 Intel Quartus Prime project file, the Platform Designer-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 37. 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. Intel 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 Intel-supplied create-this-app scripts that are included in the embedded design examples on the All Design Examples web page of the Intel 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 Platform Designer. 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 Platform Designer, 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, Intel 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:

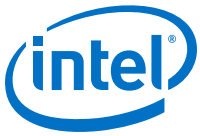
1. **Hardware project hand-off**—The hardware group provides copies of the <*system_name*>**.sopcinfo** and <*system_name*>**.sof** files. The software group copies these files to the software group's hardware project folder.
2. **Recreate software project**—The software team recreates the software application 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

- Design Examples in the Intel Design Store



- Configuring the BSP Project on page 146

4.2.2.5.2. 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 Platform Designer, 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 Platform Designer 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

Related Information

Embedded Peripherals IP User Guide

4.2.3. 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.

4.2.3.1. 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.

4.2.3.1.1. 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", Intel 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 146

4.2.3.1.2. 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 182

4.2.3.1.3. 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 164

4.2.3.1.4. 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 459

4.2.3.1.5. Configuring Peripherals and Services

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

Related Information

HAL Peripheral Services on page 165

4.2.3.2. 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.

4.2.3.2.1. 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 Platform Designer 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.

4.2.3.2.2. 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 Intel-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: Platform Designer 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 zeros. 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 *<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 161
- HAL Linking Behavior on page 182
- Software Debugging in Nios II Software Build Tools for Eclipse on page 156

4.2.3.2.3. 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 50 on page 465.

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 163
- Selecting the Operating System (HAL versus MicroC/OS-II RTOS) on page 146
- Character Mode Devices on page 168
- Software Application Optimization on page 459

4.2.3.3. 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 Intel-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

4.2.3.3.1. 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 Intel 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 Platform Designer 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 Platform Designer design as the system clock timer.

Intel 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 Intel 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 Platform Designer.
- **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 occurs 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 Platform Designer 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 Platform Designer design as the timestamp timer.

Intel 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 Platform Designer system. You can modify this frequency in Platform Designer.
- **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/*timer frequency*) × 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

4.2.3.3.2. 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, Intel 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`.

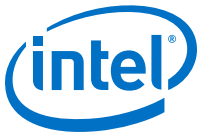
Make sure that you assign values individually for each of the `stdin`, `stdout`, and `stderr` file variables that you use.

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 155

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 18. 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 19. 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

4.2.3.3.3. Flash Memory Devices

The HAL BSP library supports parallel common flash interface (CFI) memory devices and Intel 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 Intel 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 Intel-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 176

4.2.3.3.4. 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 Platform Designer 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 464

4.2.3.3.5. 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 Intel website, and look for **Intel FPGA 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 459
- 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 459
- Developing Programs Using the Hardware Abstraction Layer
- Read-Only Zip File System

4.2.3.3.6. Unsupported Devices

The HAL provides a wide variety of native device support for Intel-supplied peripherals. However, your system may require a device or peripheral that Intel does not provide. In this case, one or both of the following two options may be available to you:

- Obtain a device through Intel's third-party program
- Incorporate your own device

Intel's third-party program information is available on the Nios II embedded software partners page. Refer to the Nios II Processor page of the Intel website, and look for **Intel FPGA 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 Gen2 Hardware Development Tutorial

4.2.3.4. 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.

4.2.3.4.1. 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.

4.2.3.4.2. Accessing Peripherals

If your application accesses peripheral registers, or performs only a small set of memory accesses, Intel 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 20. 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 21. 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: Intel recommends that you use the HAL-supplied macros for accessing external peripherals and memory.

4.2.3.4.3. 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.

4.2.3.4.4. 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 22. 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 23. 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. Intel recommends that you flush from the cache only the entries for the memory region that you make available to the other master peripheral.

4.2.3.5. 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 Platform Designer.
- **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, Intel 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
- Exception Handling

4.2.3.6. 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 Platform Designer.
- **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 463

4.2.4. 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.

4.2.4.1. 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.

4.2.4.2. 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.

4.2.4.3. HAL Linking Behavior

This section describes the default linking behavior of the BSP generation tools and how to control the linking explicitly.

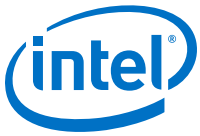
4.2.4.3.1. Default BSP Linking

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 Intel-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.



4.2.4.3.2. 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

4.3. 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 Intel 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.

4.3.1. Requirements

To use this section effectively, you need to be familiar with 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*.

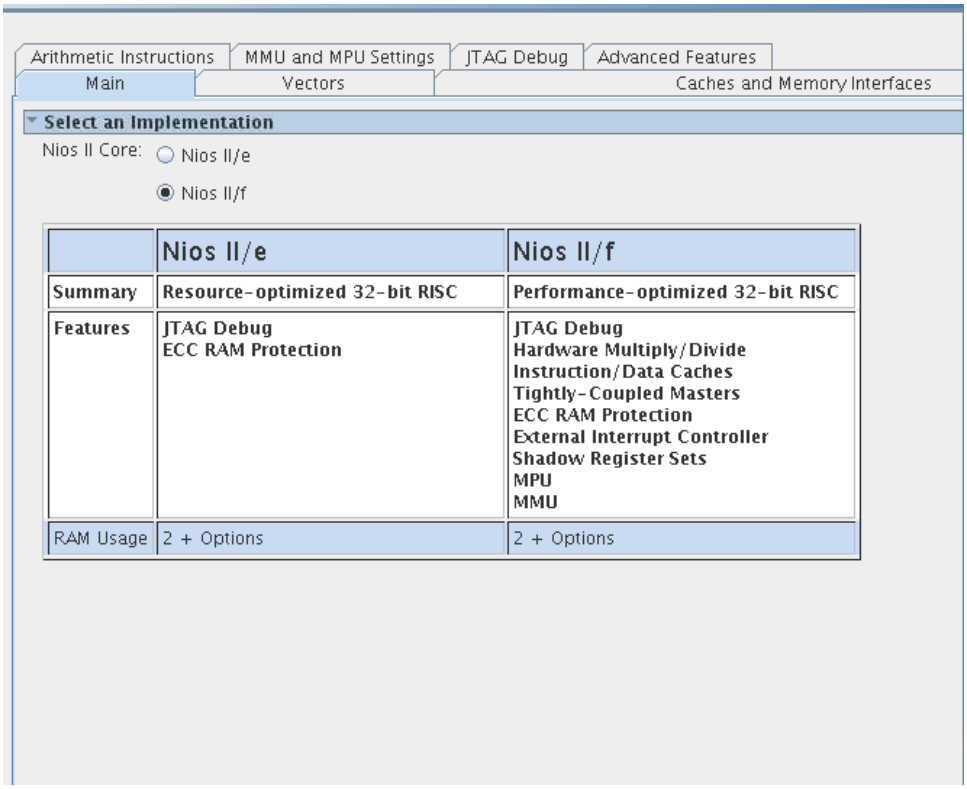
4.3.2. General Usage

This section describes the process of configuring the Nios II MPU hardware and writing software to support it.

4.3.2.1. Adding the MPU Hardware

To add an MPU to your system, you must use a Nios II/f core. In Platform Designer, 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 38. 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 39. MMU and MPU Settings Tab

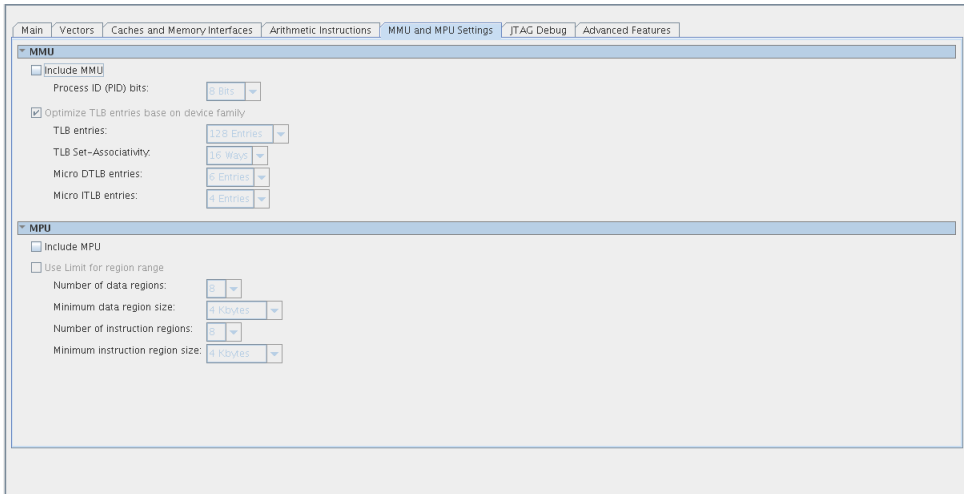


Table 17. MPU Configuration Options

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 187

4.3.2.2. Writing Software for the MPU

This section describes the process of writing software to configure and manage the Nios II MPU.

4.3.2.2.1. 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 Intel FPGA 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 191
- Operating Systems and the MPU on page 187

4.3.2.2.2. 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.

4.3.2.2.3. 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 18. mpubase Control Register Fields

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BASE[3]															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BASE[3]								0		INDEX[2]					D

Table 19. mpubase Control Register Field Descriptions

Field	Description	Access	Reset	Available
BASE	BASE is the base memory address of the region identified by the INDEX and D fields.	Read/Write	0	Only with MPU
INDEX	INDEX is the region index number.	Read/Write	0	Only with MPU
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	Only with MPU

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, ...

Equation 1. Base Address Computation

BASE = <base address>/<min_region>

For example, if the region starts at 0x1000 and the minimum region size is 256 bytes, set the BASE field to 0x10, which is 0x1000/256.

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.

(2) This field size is variable. Unused upper bits must be written as zero.

(3) This field size is variable. Unused upper bits and unused lower bits must be written as zero.

Register mpuacc Usage

mpuacc has two possible layouts, depending on the Platform Designer 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.

Table 20. mpuacc Control Register Fields for MASK Variation

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MASK[n-1:p][4]															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MASK[n-1:p][4]								0	MT		PERM			RD	WR

Table 21. mpuacc Control Register Fields for LIMIT Variation

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LIMIT[n:p][4]															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LIMIT[n:p][4]								0	MT		PERM			RD	WR

Table 22. mpuacc Control Register Field Descriptions

Field	Description	Access	Reset	Available
MASK	MASK specifies the size of the region.	Read/Write	0	Only with MPU
LIMIT	LIMIT specifies the upper address limit of the region.	Read/Write	0	Only with MPU
MT	(MT) Memory Type: • 0 = peripheral (non-cacheable, non-write bufferable) • 1 = normal (cacheable, write bufferable) • 2 = device (non-cacheable, write bufferable) • 3 = reserved	Read/Write	0	Only with MPU
PERM	PERM specifies the access permissions for the region.	Read/Write	0	Only with MPU
RD	RD is the read region flag. When RD = 1, wrctl instructions to the mpuacc register perform a read operation.	Write	0	Only with MPU
WR	WR is the write region flag. When WR = 1, wrctl instructions to the mpuacc register perform a write operation.	Write	0	Only with MPU

(4) This field size is variable. Unused upper bits and unused lower bits must be written as zero.

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.

Equation 2. Computing Region Mask

$$MASK = 0xFFFFFF << \log_2 (<region_size> >> 8)$$

The table below lists every possible MASK value for an MPU configured with a256-byte minimum region size.

Table 23. MASK Region Size Encodings

MASK Encoding	Region Size
0xFFFFFF	256 bytes
0xFFFFFE	512 bytes
0xFFFFFC	1 KB
0xFFFFF8	2 KB
0xFFFFF0	4 KB
0xFFFFE0	8 KB
0xFFFFC0	16 KB
0xFFFF80	32 KB
0xFFFF00	64 KB
0xFFFE00	128 KB
0xFFFC00	256 KB
0xFFF800	512 KB
0xFFF000	1 MB
0xFFE000	2 MB
0xFFC000	4 MB
0xFF8000	8 MB
0xFF0000	16 MB
0xFE0000	32 MB
0xFC0000	64 MB
0xF80000	128 MB
0xF00000	256 MB
0xE00000	512 MB
0xC00000	1 GB
0x800000	2 GB
0x000000	4 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 256 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 0x10, which is 0x1000/256 or (0x1000 >> 8)
- Set mpuacc.LIMIT to 0x20, which is 0x2000/256 or (0x2000 >> 8)

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 24. Instruction Region Permission Encodings

PERM Encoding	Supervisor Permissions	User Permissions
000	None	None
001	Execute	None
010	Execute	Execute

Table 25. 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 184

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 0x100000000, which is 2^{32} (the size of the Nios II address space). For example, if the minimum region size is 256, or 0x100 bytes, `mpuacc.MASK` is 0x100000000 / 0x100, or 0x10000000.
- If the MPU is configured to define region size by limit, `mpuacc.LIMIT = 0`.

4.3.2.2.4. 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.

4.3.3. 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 Intel FPGA HAL.

This design example is based on Nios II classic processor. For more information about MPU register, refer to Nios II Classic Processor Reference Guide.

4.3.3.1. Working with the MPU Design Examples

This section's design examples and software examples are developed for, and tested with, the following:

- The Nios II Embedded Evaluation Kit (NEEK), Cyclone® III Edition

  Note: The NEEK is no longer available. However, the design examples use only on-chip hardware resources. Therefore, it is easy to port the designs to a different hardware platform.
- Quartus software version 9.1.

You also need the following:

- Nios II Embedded Design Suite (EDS).
- The design example archive file, `an540_91.zip`.

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.

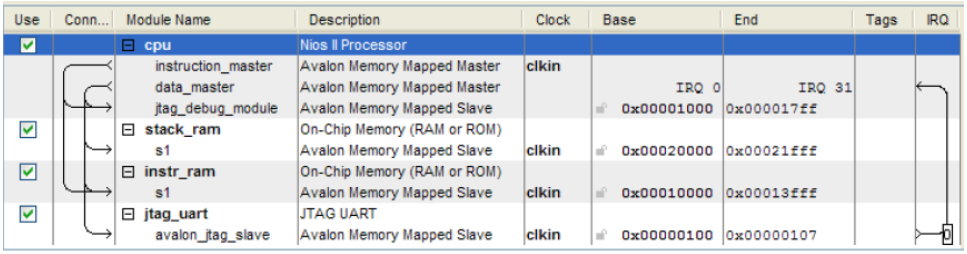
4.3.3.2. Example Hardware

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 41 on page 194.

The figure below illustrates one of the design examples as it appears in Platform Designer. The hardware addresses fall on valid MPU region boundaries. While this constraint is not required, it is more convenient for the software engineer.

Figure 40. MPU Example Hardware System

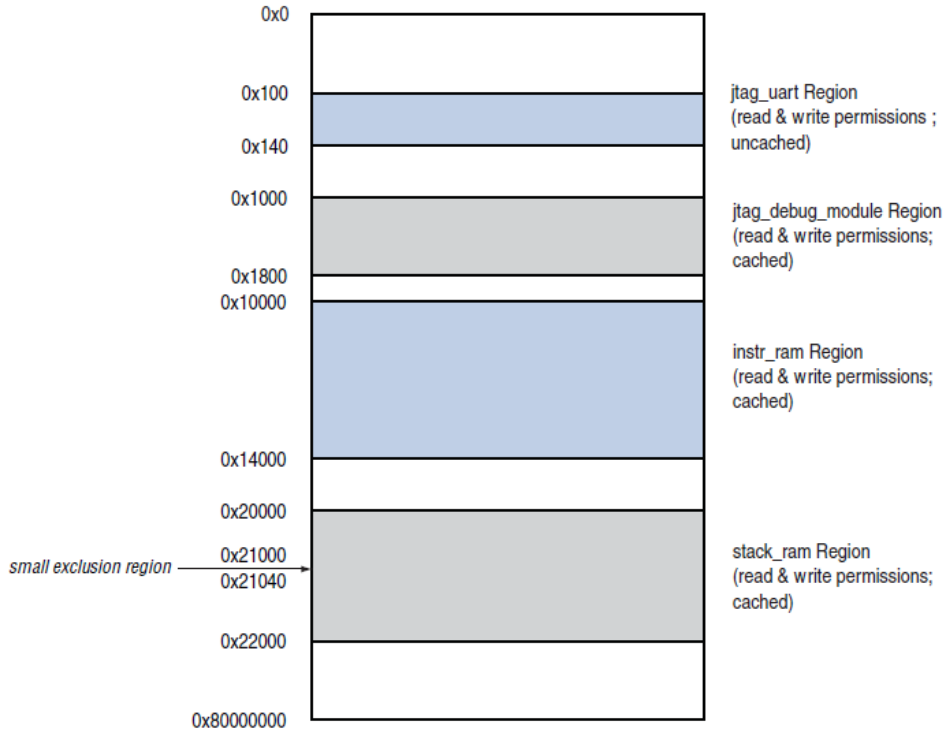


4.3.3.3. Region Layout Considerations

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.

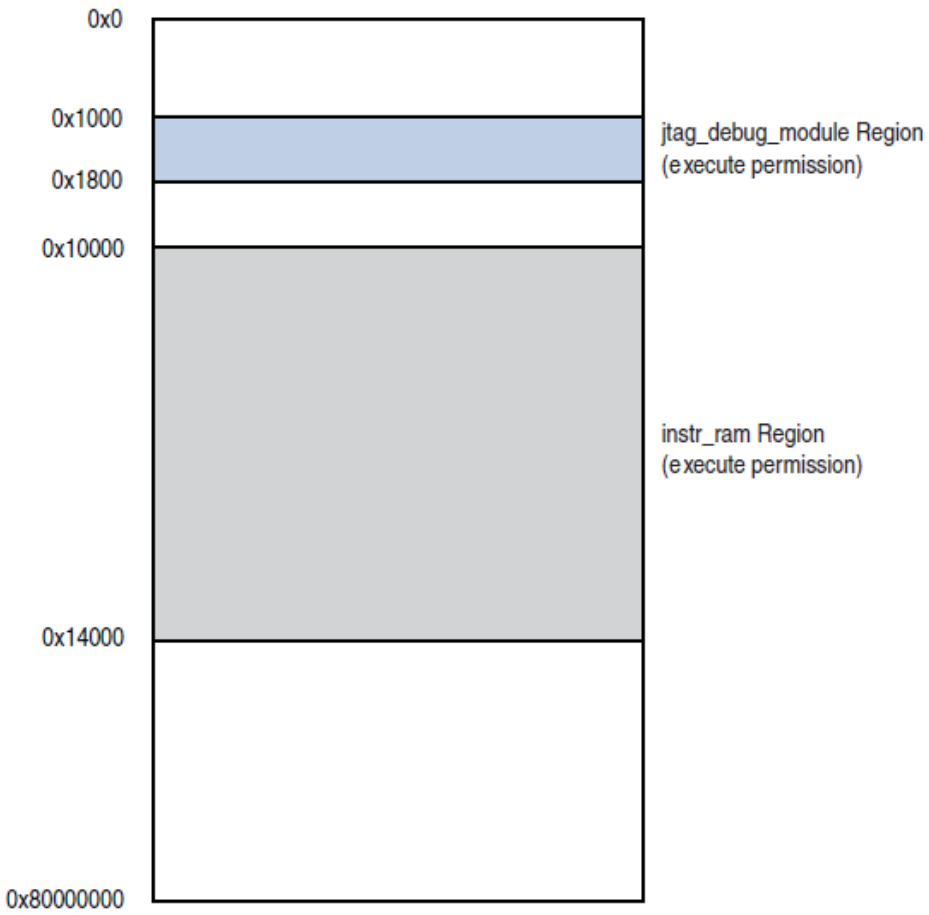
Figure 41. MPU Data Region Example (Addresses not to scale)



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 42. 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 187

4.3.3.4. 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.

4.3.3.4.1. 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 24. 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 nios2_mpu_inst_init() in the mpu_basic Software Example on page 197, `region[0]` grants execution access to the **instr_ram** memory in both user and supervisor modes, as shown in Figure 42 on page 195. `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 25. 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 = 0x1fffe0;
#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 26. 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.

4.3.3.4.2. 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 > Intel FPGA > 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 27. mpu_basic Console Output

```
Using cable "Intel FPGA Download Cable [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: "Intel FPGA Download Cable [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 200.

mpu_exc_detection Console Output

Example 28. mpu_exc_detection Console Output

```
Using cable "Intel FPGA Download Cable [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: "Intel FPGA Download Cable [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**.

4.4. Profiling Nios II Systems

This chapter describes the methods to measure the performance of a Nios II system with the GNU profiler (**nios2-elf-gprof**), the performance counter component, and the timestamp interval timer component. This chapter also includes two tutorials to measure performance in the Intel Nios II Software Build Tools (SBT) development flow.

The Nios II development environment provides several tools to analyze the performance of your project. The software-only GNU profiler approach adds minimal overhead. To analyze deterministic real-time performance issues, you can use a hardware timer or a performance counter. To choose the best tool for your task, consider the problem that you are solving.

4.4.1. Requirements

You must be familiar with the Nios II SBT development flow for Nios II systems, including the Quartus Prime software and Platform Designer to use the tutorials.

4.4.1.1. Obtaining the Hardware Design

The tutorials in this chapter work with the Nios II Ethernet Standard Design Example.

To use the design example, unzip the **.zip** for your development kit to a working directory on your system.

Note: This chapter refers the software example directory as <*project_directory*>.

Related Information

Profiling Nios II Systems Design Example Webpage

4.4.1.2. Obtaining the Software Examples

To obtain the software examples for this chapter, follow these steps:

1. Download the profiler_software_examples.zip.
2. Unzip the **profiler_software_examples.zip** to <*project_directory*> in your system.

Note: This chapter refers the directory as <*profiler_software_examples*>.

Related Information

Profiling Nios II Systems Design Example Webpage

4.4.2. Tools

You can use the GNU profiler without making any hardware changes to your Nios II system. This tool directs the compiler to add calls to profiler library functions into your application code.

The performance counter component and the timestamp component are minimally intrusive hardware methods for measuring the performance of a Nios II system. This chapter describes and compares the two components. To use these methods, you add

the hardware components to your system, and you add macro invocations to your source code to start and stop the components. The hardware components perform the measurements.

Compiler speed optimizations affect functions to widely varying degrees. Compiler size optimizations also affect functions in different ways. These differences impact the cache usage and the resource contention, which can change the relative start times and therefore increase the execution times of functions. For these reasons, you must optimize your code with the -O3 compiler switch, and then perform profiling on the code to gain the most insight on how to improve an application in its final form.

The tutorials use three tools to measure the performance of a Nios II system, as described in the following sections:

- GNU Profiler
- Intel Performance Counter
- High-Resolution Timer

In addition, the program counter trace collection tool is available for some Nios II processors. However, the tutorials do not use this tool.

You use the GNU profiler to identify the areas of code that consume the most CPU time, and a performance counter or a timer component to analyze functional bottlenecks.

4.4.2.1. GNU Profiler

You must make minimal changes to the source code to take measurements for analysis with the GNU profiler. To implement the required changes, follow these steps:

1. In the Nios II SBT, enable the GNU profiler in your project by turning on the `hal.enable_gprof` and `hal.enable_exit` board support package (BSP) settings.

   Note: If you use the Nios II SBT for Eclipse, the software enables `hal.enable_exit` by default.

2. Verify that your `main()` function returns.

   Note: When `main()` calls `return()` or terminates, `alt_main()` calls `exit()` as appropriate for profiling. The `exit()` function runs the BREAK 2 instruction, which causes the profiling data to write to the **gmon.out** on the host computer.

3. Rebuild the BSP and the application project.

4.4.2.2. Intel Performance Counter

A performance counter is a block of counters in the hardware that measures the execution time of the code sections that you choose. A performance counter component can track up to seven code sections. By default, the component tracks three code sections. A pair of counters tracks each code section:

- Time—A 64-bit time (clock tick) counter that counts the number of clock ticks during code section runs.
- Occurrences—A 32-bit event counter that counts the number of times the code section runs.

Note: You can change the maximum number of measured code sections by editing the performance counter component in Platform Designer.

These counters enable you to measure the execution time of the designated sections of C/C++ code. Macros enable you to mark the start and the end of the code sections in your program. The performance counter component has up to seven pairs of counters, supporting as many as seven measured sections of C/C++ code. You must add macros to your code at the start and end of each measured section. An additional, built-in pair of counters aggregates the individual code section counters, enabling you to measure each section as a fraction of a larger program.

You can use performance counters for analyzing determinism and other runtime issues.

Note: The performance counter component occupies a substantial number of logic elements (LEs) on your device, and requires software implementation to obtain performance measurements.

4.4.2.3. High-Resolution Timer

A high-resolution timer, in contrast to a performance counter component, does not use a large number of LEs on your device, and does not require heavy implementation of every function call in your code to obtain performance measurements. Timers require explicit read calls in the sections of the source code that you want to measure, so their use is better suited for pinpointing the performance issues in a program. You must implement the source code manually; however, because this implementation is less pervasive, therefore, this implementation is also less intrusive. Unlike the performance counter macros, the timer requires many more processor cycles to make two function calls; one to read the time at the beginning of a measured section, and one to read the time at the end.

4.4.2.4. Program Counter Trace Information

The Nios II processor can generate complete and accurate program counter trace information. However, the GNU profiler does not use this information. To generate this information, you must have a Nios II processor configured with a JTAG debug module of level 3 or greater. The level 3 JTAG debug module creates on-chip trace data. You can capture approximately a dozen instructions in the on-chip trace buffer.

You can obtain a much larger trace by configuring a Nios II core with a level 4 JTAG debug module to generate off-chip trace information; however, you need a First Silicon Solutions, Inc. (FS2) or Lauterbach Datentechnik GmbH (Lauterbach) (www.lauterbach.com) hardware to collect this off-chip trace data.

For more information about the Lauterbach hardware, refer to the “Debuggers” section in the Debugging Nios II Designs chapter.

4.4.3. Using the GNU Profiler to Measure Code Performance

The following sections explain the advantages and limitations of using the GNU profiler for performance analysis. A tutorial demonstrates the use of the GNU profiler to collect and analyze performance data.

4.4.3.1. GNU Profiler Advantages

The major advantage to measuring performance with the GNU profiler is that the GNU profiler provides an overview of the entire system. Although the GNU profiler adds some overhead, the GNU profiler distributes this overhead throughout the system evenly. The functions the GNU profiler identifies as consuming the most processor time also consume the most processor time when you run the application at full speed without profiler implementation.

4.4.3.2. GNU Profiler Limitations

Adding instructions to each function call for use by the GNU profiler affects the behavior of the code in the following ways:

- Each function is slightly larger due to the additional function call to collect profiling information.
- The entry and the exit time of each function due to profiling information collection.
- The instruction-cache misses are higher because the profiling function is in the instruction cache memory.
- Memory that records the profiling data can change the behavior of the data cache.

These effects can mask the time sensitive issues that you are trying to uncover through profiling.

The GNU profiler determines the percentage of time spent in each function by interpolation, based on periodic samplings of the program counter. The GNU profiler ties the periodic samples to the timer tick of the system clock. The GNU profiler can take samples only when you enable interrupts, and therefore cannot record the processor cycles spent in interrupt routines.

The GNU profiler cannot profile individual functions. You can use the GNU profiler to profile the entire system, or not at all.

The profiling data is a sampling of the program counter at the resolution of the system timer tick. Therefore, the profiling data provides estimation, not an exact representation, of the processor time spent in different functions. You can improve the statistical significance of the sampling by increasing the frequency of the system timer tick. However, increasing the frequency of the tick increases the time spent recording samples, which in turn affects the integrity of the measurement.

Note: To use the GNU profiler successfully with your custom hardware design, you must ensure that your design includes a system clock timer. The GNU profiler requires this component to produce proper output.

4.4.3.3. Software Considerations

The GNU profiler implements your source code with functions to track processor usage.

4.4.3.3.1. Profiler Mechanics

You enable the GNU profiler by turning on the hal.enable_gprof switch in the scripts to generate the BSP. Turning on this switch automatically turns on the -pg compiler switch and then links the profiling library code in the altera_nios2 software component with the BSP. This code counts the number of calls to each profiled function.

The `-pg` compiler option forces the compiler to insert a call to the `mcount()` function (located in the file `altera_nios2/HAL/src/alt_mcount.S`) at the beginning of every function call. The calls to `mcount()` track every dynamic parent and child function call relationship to enable the construction of a call graph. The option also installs `nios2_pcsample()function` (located in the file `altera_nios2/HAL/src/alt_gmon.c`) that samples the foreground program counter at every system clock interrupt. When the program executes, the GNU profiler collects data on the host of the **gmon.out**. The **nios2-elf-gprof** utility can read this file and display profiling information about the program.

The profiling code operates on the target by performing the following steps:

1. The Compiler implements function prologues with a call to `mcount()` to enable the Compiler to determine the function call graph. The GNU profiler documentation refers to this data as the function call arc data.
2. The timer interrupt handler registers an alarm to capture information about the foreground function (histogram data) that executes when the alarm triggers.
3. The heap allocates a target memory to store the profiling data.
4. When your code exits with a BREAK 2 instruction, the **nios2-download** utility copies the profiling data from the target to the host.

The **nios2-elf-gprof** utility requires the function call arc data and the histogram data to work correctly.

Note: For more information about the GNU profiler, refer to the Nios II GNU profiler documentation, included with the GCC documentation, available on the Nios II Embedded Design Suite Support page of the Intel website.

4.4.3.3.2. Profiler Overhead

Using the GNU profiler impacts memory and processor cycles.

Memory

The impact of the profiling information on the `.text` section size is proportional to the number of small functions in the application. The code overhead—the size of the `.text` section—increases when the GNU profiler enables profiling, due to the addition of the `nios2_pcsample()` and `mcount()` functions. The GNU profiler implements the system timer with a call to `nios2_pcsample()`, and implements every function with a call to `mcount()`. The `.text` section increases by the additional function calls and by the sizes of these two functions.

To view the impact on the .text section, you can compare the sizes of the `.text` sections in the **.objdump**.

The GNU profiler uses buckets to store data on the heap during profiling. Each bucket is two bytes in size. Each bucket holds samples for 32 bytes of code in the .text section. The total number of profiler buckets allocated from the heap is when you divide the size of the .text section by 32. The heap memory that the GNU profiler buckets consume is therefore:

((.text section size) / 32) × 2 bytes

The GNU profiler measures all functions in the object code that the GNU profiler compiles with profiling information. This set of functions includes the library functions, which include the run-time library and the BSP.

Processor Cycles

The GNU profiler tracks each individual function with a call to `mcount()`. Therefore, if the application code contains many small functions, the impact of the GNU profiler on processor time is larger. However, the resolution of the profiled data is higher. To calculate the additional processor time consumed by profiling with `mcount()`, multiply the amount of time that the processor requires to execute `mcount()` by the number of run-time function calls in your application run.

On every clock tick, the processor calls the `nios2_pcsample()` function. To calculate the required additional processor time to perform profiling with `nios2_pcsample()`, multiply the time the processor requires to execute this function by the number of clock ticks that your application requires, which includes the time the `mcount()` calls and execution requires.

To calculate the number of additional processor cycles used for profiling, add the overhead you calculated for all the calls to `mcount()` to the overhead you calculated for all the calls to `nios2_pcsample()`.

4.4.3.4. Hardware Considerations

The GNU profiler requires only a system timer. If your Nios II hardware design includes a system timer, you do not need to change your design.

4.4.3.5. Tutorial: Using the GNU Profiler

For demonstration purposes, this tutorial uses the Nios II Ethernet Standard design example for the Nios II Embedded Evaluation Kit, Cyclone III Edition (NEEK) development kit. You could use other similar design examples which target your development kit.

To configure your device with the design example, follow these steps:

1. Start the Intel Quartus Prime software.
2. On the File menu, click **Open Project**.
3. Open **niosii_ethernet_standard_3c25.qpf**.
4. On the Tools menu, click **Programmer**.
5. Click **Start** to download the SRAM Object File (**.sof**) to your device.

Note: If the software disabled the **Start** button, or the **Hardware Setup** field does not list the Intel FPGA Download Cable, refer to the Introduction to the Quartus Software manual for more details on the Programmer tool.

4.4.3.5.1. Profiler Example with the Nios II Command Line

Creating the Profiler Software Example

To create the profiler_gnu software project in the Nios II command-line flow, follow these 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 > Intel FPGA > Nios II EDS` <*version*>, and click **Nios II** <*version*> **Command Shell**.
   - In the Linux operating system, in a command shell, change directories to <*Nios II EDS install path*>, and type the command *./sdk_shell*.
2. Change to the directory <*profiler_software_examples*>`/app/profiler_gnu`
3. Create and build the application with the **create-this-app** script, by typing the following command:

   ```
   ./create-this-app
   ```

   The **create-this-app** script runs the **create-this-bsp** script, which reads settings from the **parameter_definition.tcl** in <*profiler_software_examples*>`/bsp/hal_profiler_gnu`. This Tcl file contains the following lines:

   ```
   set_setting hal.enable_gprof true
   ```

   ```
   set_setting hal.enable_exit true
   ```

   The first setting enables the GNU profiler, and the second setting enables the `alt_main()` function to call `exit()` following `main()`.

Running the Profiler Software Example

To run the application and collect the GNU profiler data, follow these steps:

1. Open a second Nios II command shell.
2. In the second shell, open a **nios2-terminal** session by typing the following command:

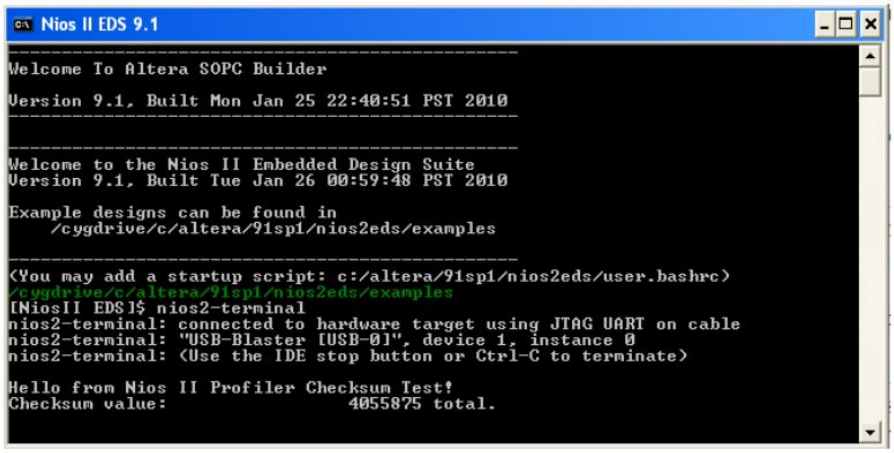
   ```
   nios2-terminal
   ```

3. In your original Nios II command shell, download the .elf to the development board, run your design, and write the GNU profiler data to the gmon.out, by typing the following command:

   ```
   nios2-download -g --write-gmon gmon.out *.elf
   ```

   The GNU profiler collects data while the application runs, and then writes the data to the **gmon.out** when the application calls the `exit()` function. The figure below shows an example of the GNU profiler output in the Nios II command shell.
4. **Exit nios2-terminal** by typing `control-C`.

Figure 43. GNU Profiler Output on Nios II Command Shell



Creating the GNU Profiler Report

When you run your project, your project creates the gmon.out. You must format this file to a readable format. To format this file, follow these steps:

1. In the original Nios II command shell, change your directory to *<profiler_software_examples>***/app/profiler_gnu**.
2. Type the following command:

   ```
   nios2-elf-gprof profiler_gnu.elf gmon.out > report.txt
   ```

3. Use any text editor to view the **report.txt**.

   For more information about the GNU profiler report, refer to "Analyzing the GNU Profiler Report".

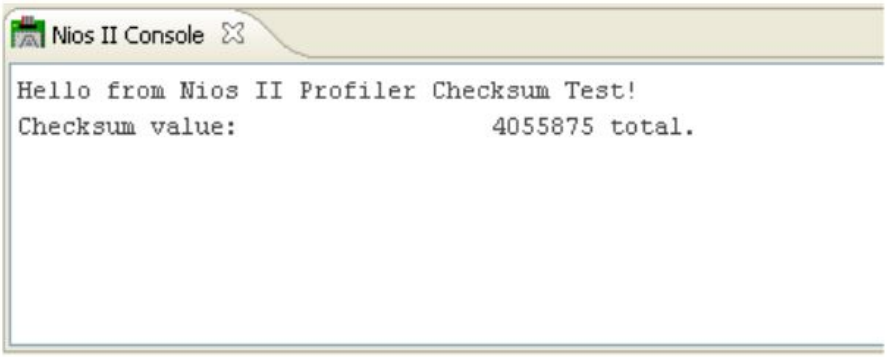
4.4.3.5.2. Profiler Example with Nios II SBT for Eclipse

Creating and Running the Profiler Software Example

1. Start the Nios II SBT for Eclipse.
2. Under File menu, point to **New**, and click **Nios II Application and BSP from template**.
3. Set **Platform Designer Information File name** by browsing to locate the Platform Designer Information File (**.sopcinfo**) in *<project_directory>*.
4. Name your project, such as profiler_gnu.
5. Under **Templates**, select **Blank Project**.
6. Click **Finish** to create your new project.
7. Locate the *<profiler_software_examples>***/eclipse_source_files/profiler_gnu** folder and copy all the files in this directory. In Nios II SBT for Eclipse, right click on **profiler_gnu** in Project Explorer view and select **Paste**.
8. Right click your project in the Project Explorer view, point to **Nios II** and click **BSP Editor**.

9. In the Nios II BSP Editor, turn on `hal.enable_gprof` to enable the GNU profiler in your project.
10. Generate your BSP project and exit.
11. Right click your project in the Project Explorer view and then click **Build Project**.
12. To download and run the profiler_gnu software, right click your project, point to **Run As**, and then click **Nios II Hardware**.

Figure 44. Nios II Console After Running profiler_gnu



Viewing the GNU Profiler Report

The software creates the gmon.out in your project folder, which you can view in the Project Explorer view of the Nios II SBT for Eclipse. If the **gmon.out** does not appear, right click on your project and select **Refresh**. When you open **gmon.out**, the Nios II SBT for Eclipse switches to the Profiling view, in which you can view the report. For more information about the GNU profiler report, refer to "Analyzing the GNU Profiler Report".

4.4.3.5.3. Analyzing the GNU Profiler Report

The information in this section is applicable to the GNU profiler report that the command line or the Nios II SBT for Eclipse generates.

The GNU profiler report contains information in the following formats:

- The **flat profile** portion of the report identifies the child functions in the order in which they consume processing time.
- The **call graph** portion of the report describes the call tree of the program sorted by the total amount of time spent in each function and its children. Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called, with exceptions and conditions detailed further in the report itself and the GNU profiler documentation.

Note: For more information, refer to the Nios II GNU profiler documentation, with the GCC documentation, available at the Nios II Embedded Design Suite Support page.

The example below shows the GNU profiler report excerpts from the previous tutorial. In the example, the flat profile shows that the **checksum_test_routine()** function call consumed 79.19% of the processing time during the execution.

The granularity statement in the call graph report states that the report covers 2.55 seconds (2550 milliseconds). The Nios II timer (`sys_clk_timer`) has a 10 millisecond timer. The GNU profiler calls the timer interrupt once at the beginning, before a full clock period elapsed, and once every 10 milliseconds thereafter. A precise report, therefore, would show that the GNU profiler calls the timer interrupt handler 255 times. Index[13] shows that the GNU profiler calls `alt_avalon_timer_sc_irq()` 256 times, which is in the precision range of this measurement method.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative    self           self     total
time    seconds   seconds   calls  s/call   s/call   name
79.19       2.02     2.02       1    2.02    2.03    checksum_test_routine
18.01       2.48     0.46       1    0.46    0.46    alt_busy_sleep
    .
    .
    .
        Call graph (explanation follows)

granularity: each sample hit covers 32 byte(s) for 0.39% of 2.55 seconds

index     % time    self  children called   name
            0.00   0.00       273/273      alt_irq_entry [106]
[13]   0.0    0.00   0.00          273       alt_irq_handler [13]
            0.00   0.00       256/256      alt_avalon_timer_sc_irq [14]
            0.00   0.00        17/17       altera_avalon_jtag_uart_irq [17]
    .
    .
    .
```

4.4.4. Using Performance Counter and Timer Components

After the GNU profiler identifies areas of code that consume the most processor time, a performance counter or a timer component can further analyze these functional bottlenecks.

The following sections describe the advantages and limitations of using performance counters and timers for performance analysis. A tutorial demonstrates the use of performance counters and timers to collect and analyze performance data.

4.4.4.1. Performance Counter Advantages

Performance counters are the only mechanism available with the Nios II development kits that provide measurements with little intrusion. You can use efficient macros to start and stop the measurement for each measured section. A performance counter is an order of magnitude faster than the GNU profiler. The only less intrusive way to collect measurement data would be a completely hardware-based solution, such as a logic analyzer configured with triggers on particular bus addresses.

4.4.4.2. Timer Advantages

Unlike the performance counter, which can track only seven sections of code simultaneously, the timer has no such limit. You can read the timer 1,000 times and store the timer in 1,000 different variables as a start time for a section. Then, you can compare the timer to 1,000 end timer readings. The only practical limiting factors are memory consumption, processor overhead, and complexity.

4.4.4.3. Performance Counter and Timer Hardware Considerations

One disadvantage to measuring performance with a performance counter is the size of the counter. The performance counter component consumes a large number of LEs on your device.

On a 3C120 device, a single performance counter component with three section counters defined in a modified standard hardware design consumes 671 logic cells (LCs), and 420 LC registers. In the same design, a single performance counter defined with seven section counters consumes 1339 logic cells and 808 LC registers. The resource usage of the performance counter component is nearly identical on all devices.

Note: Remove the performance counter from the final version of your system to save resources.

The timer consumes hardware resources, although substantially less than a performance counter. The timer also introduces an additional interrupt source in the system that impacts interrupt latency.

Note: Adding performance counters and timers can reduce f_{MAX}.

4.4.4.4. Performance Counter and Timer Software Considerations

A common disadvantage of performance counters and timers is the lack of context awareness. If a timer interrupt occurs during the measurement of a section of code, performance counters and timers add the time taken by the processor to process the timer interrupt to the total measurement time. This effect occurs for simple interrupts and multithreading context switching, although this effect occurs more in a multithreaded system. Many threads or interrupt service routines might execute while you measure the section of code, resulting in a very large, skewed measurement. The resulting measurement distortion is unpredictable, and has no correlation with the behavior of the code section you are attempting to measure.

To avoid context switch impacts, most multithreaded operating systems have a system call to temporarily lock the scheduler. Alternatively, you can disable interrupts to avoid context switches during section measurement.

Note: Disabling interrupts or locking the scheduler affects the behavior of your system, so you must use these techniques only as a last resort.

4.4.4.5. Performance Counter Software Considerations

You must use the `PERF_BEGIN` and `PERF_END` performance counter macros to record the beginning and ending times of each measured section.

`PERF_BEGIN` and `PERF_END` are single writes to the performance counter component. These macros are very efficient, requiring only two or three machine instructions.

The example below shows the PERF_BEGIN and PERF_END performance counter macros in **altera_avalon_performance_counter.h**:

Example 29. PERF_BEGIN and PERF_END Performance Counter Macros in altera_avalon_performance_counter.h

```
#define PERF_BEGIN(p,n) IOWR((p),(((n)*4)+1),0)

#define PERF_END(p,n) IOWR((p),(((n)*4) ),0)
```

4.4.4.6. The Global Counter

The performance counter component contains several counters. You can configure the number of measured sections in Platform Designer. You have one pair of counters for each measured section, as described in "Intel FPGA Performance Counter" section. In addition, the performance counter component always has a global counter.

The global counter measures the total time of the measurement. When you stop the global counter, other counters do not run. The PERF_START_MEASURING and PERF_STOP_MEASURING macros control the global counter.

Warning: Do not attempt to manipulate the global counter in any other way.

For more information about performance counters, refer to the Performance Counter Core chapter in the *Embedded Peripherals IP User Guide*.

4.4.4.7. Hardware Considerations

Performance counters and timestamp interval timers are Platform Designer components. When you add one to an existing system, you must regenerate the Platform Designer system and recompile the .sof in the Intel Quartus Prime software. Timers and performance counters can eventually overflow, such as any hardware counter.

4.4.4.8. Tutorial: Using Performance Counters and Timers

This tutorial demonstrates the use of performance counters and timestamp interval timers to measure the performance of a Nios II system more precisely than is possible with the GNU profiler, by identifying the sections of code that use the most processor time.

This tutorial uses the same NEEK design as the GNU profiler tutorial. This design has an interval timer and a performance counter. You can change the timer interval and the number of sections that the performance counter measures.

4.4.4.8.1. Modifying the Nios II Hardware Design

You must modify the Nios II Ethernet Standard design example for this tutorial. To modify the Nios II Ethernet Standard design example, follow these steps:

1. In Intel Quartus Prime software, on the Tools menu, click **Platform Designer**.
2. In <*project_directory*>, click **peripheral_system.qsys**.
3. Right click the **high_res_timer** module and then click **Edit**.
4. Under **Timeout period**, set the interval time **Period** to 1 and the units to **us** (microseconds).

5. Click **Finish**.
6. On the File menu, click **Save**.
7. The Nios II Ethernet Standard design example is a hierarchal based design. To generate the system, on the File menu, click **Open**, and then select **eth_std_main_system.qsys**.
8. Click the **Generation** tab.
9. Turn on the **Create HDL design files for synthesis and Create block symbol file** (**.bsf**) options.
10. Ensure that the Output Directory path is *<project_directory>***/eth_std_main_system**.
11. Click **Generate**. Save the system if the software prompts you to do so.
12. Exit Platform Designer when generation is complete.
13. To generate the **.sof**, in the Intel Quartus Prime software, on the Processing menu, click **Start Compilation**.
14. Click **OK** when the following message appears:

```
Full Compilation was successful
```

4.4.4.8.2. Programming the Hardware Design to Your Device

After compiling your modified hardware design, you can program the hardware design to your device. To do so, follow these steps:

1. On the Tools menu, click **Programmer**.
2. Click **Start** to download the **.sof** to your device.

   If the software disables the **Start** button, or the **Hardware Setup** field does not list the Intel FPGA Download Cable cable, refer to the *Introduction to the Quartus Software* manual for more details on the Programmer tool.

4.4.4.8.3. Performance Counter Example with the Nios II Command Line

This section describes how to create and run the performance counter software example with the Nios II command line.

Creating the Performance Counter Software Example

To create the **profiler_performance_counter** software project in the Nios II software build flow, follow these steps:

1. Open a Nios II command shell as described in "Creating the Profiler Software Example".
2. Change to the *<profiler_software_examples>*`/app/profiler_performance_counter` directory.
3. Create and build the application by typing the following command:

```
./create-this-app
```

The **create-this-app** script runs the **create-this-bsp** script, which reads settings from the **parameter_definition.tcl** in *<profiler_software_examples>*/bsp/hal_profiler_performance_counter. This Tcl file contains the following lines:

```
set_setting hal.sys_clk_timer peripheral_subsystem_sys_clk_timer
set_setting hal.timestamp_timer peripheral_subsystem_high_res_timer
set_setting hal.enable_gprof true
set_setting hal.enable_exit true
```

The first two lines set the system clock timer and timestamp timer to the corresponding timers in the Platform Designer system.

The third line enables the GNU profiler, and the last line enable the `alt_main()` function to call `exit()` following `main()`.

Running the Performance Counter Software Example

To run the application and collect the GNU profiler data, follow these steps:

1. Open a second Nios II command shell.
2. In the second shell, open a nios2-terminal session by typing the following command:

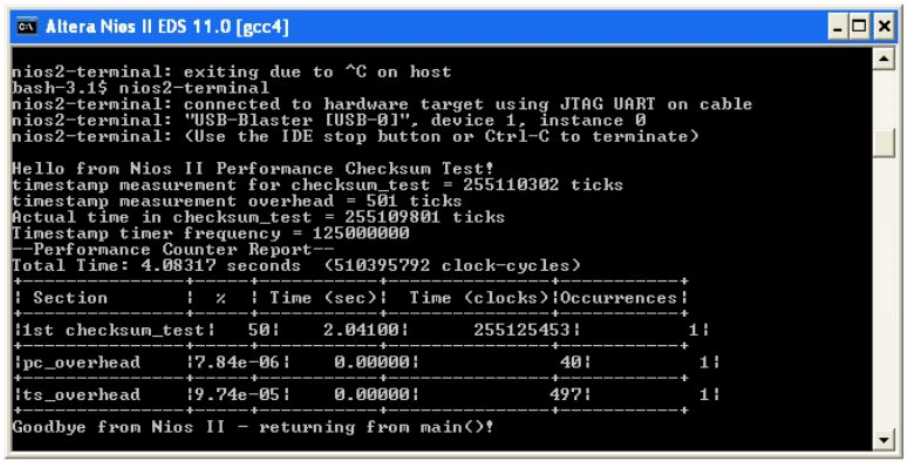
   ```
   nios2-terminal
   ```

3. In your original Nios II command shell, run the program by typing the following command:

   ```
   nios2-download -g *.elf
   ```

   The figure below shows an example of the output that appears in the Nios II command shell. Your output might vary. For more information, refer to "Analyzing the Performance Counter Report".

Figure 45. Performance Counter Report on Nios II Command Shell



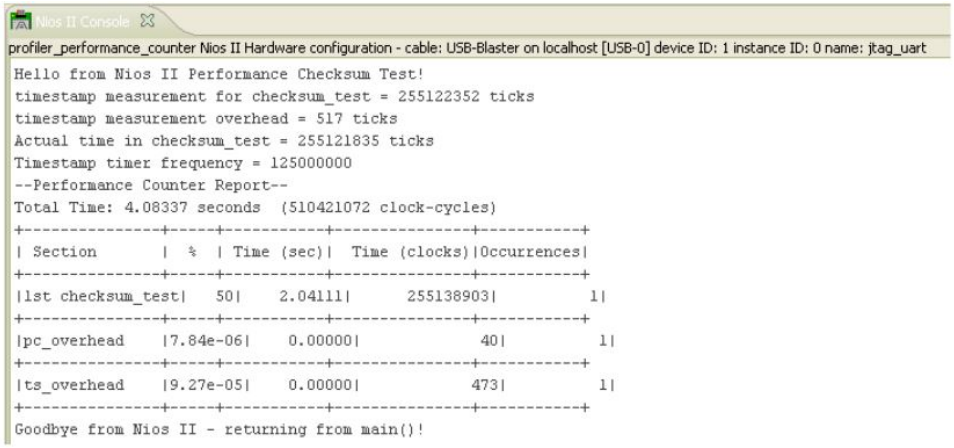
4.4.4.8.4. Performance Counter Example with Nios II SBT for Eclipse

This section describes how to create and run the profiler_performance_counter software example with the Nios II SBT for Eclipse.

1. Start the Nios II SBT for Eclipse.
2. Under File menu, point to **New**, and click **Nios II Application and BSP from template**.
3. Set **Platform Designer Information File name** by browsing to the <*project_directory*> directory and selecting the **.sopcinfo**.
4. Give your project a name, for example **profiler_performance_counter**.
5. Under **Templates**, select **Blank Project**.
6. Click **Finish** to create your new project.
7. Locate the <*profiler_software_examples*>**/eclipse_source_files/ profiler_performance_counter** folder, and copy all the files in this directory. In Nios II SBT for Eclipse, right click on profiler_gnu in Project Explorer view and select **Paste**.
8. Right click your project in the Project Explorer view, point to **Nios II** and click **BSP Editor**.
9. In the Nios II BSP Editor, turn on `hal.enable_gprof` to enable the GNU profiler in your project.
10. Set the `hal.sys_clk_timer` to the **peripheral_subsystem_sys_clk_timer** component.
11. Set `hal.timestamp_timer` to the **peripheral_subsystem_high_res_timer** component.
12. Generate your BSP project and exit.
13. Right click your project in the Project Explorer view, point to **Build Project**.
14. To run the **profiler_performance_counter** software, right click your application project, point to **Run As** and click **Nios II Hardware**.

The figure below shows the Nios II Console output after running **profiler_performance_counter**. The data are similar to the command-line example in Figure 45 on page 214. For more information, refer to "Analyzing the Performance Counter Report".

Figure 46. Performance Counter Report on Nios II Console



4.4.4.8.5. Analyzing the Performance Counter Report

The information in this section is applicable to the performance counter report that the command line or the Nios II SBT for Eclipse generates.

`pc_overhead` is the performance counter component overhead due to a single call to the `PERF_BEGIN` macro. This number includes the overhead of executing the `PERF_BEGIN` macro and the corresponding `PERF_END` macro for this measured section.

`ts_overhead` is the timestamp overhead—the overhead of a single function call to read the timer. This number includes the performance counter overhead to implement the measurement.

4.4.5. Troubleshooting

The following sections describe several problems that might occur, and suggest ways to troubleshoot the problems.

4.4.5.1. nios2-elf-gprof –annotated-source Switch Has No Effect

The profiler does not track the `basic-block-count` information, so switches (such as the `–annotated-source` switch) do not work.

4.4.5.2. Writing to the Registers of a Nonexistent Section Counter

The performance counter report in the example below shows what happens when you attempt to use a nonexistent section counter of the performance counter component.

Example 30. Result of Using a Nonexistent Section Counter

```
--Performance Counter Report--
Total Time: 5.78751 seconds (289375582 clock-cycles)
+--------------------+--------+-------------+---------------+-----------+
|     Section        |   %    | Time (sec)  | Time (clocks) |Occurrences|
+--------------------+--------+-------------+---------------+-----------+
|sleep_tests         |    49.4|      2.86162|      143081026|          1|
+--------------------+--------+-------------+---------------+-----------+
|perf_begin_overhead | 7.6e-06|      0.00000|             22|          1|
```

```
+--------------------+--------+-------------+---------------+-----------+
|timestamp_overhead  | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|non_existent_counter|6.37e+12|368934881474.19104|        -1| 4294967295|
+--------------------+--------+-------------+---------------+-----------+
```

Assume a fourth section counter specifies a performance counter component that Platform Designer defines to have three section counters only (the default value).

In the example, the test is performed on a hardware design that does not have any other component defined with registers mapped immediately after the registers of the performance counter component. Therefore, there is no impact to other component. Depending on how you configure the component register base addresses in Platform Designer for a particular hardware design, unpredictable system behavior could occur.

4.4.5.3. Output From a printf() or perf_print_formatted_output() Call Near the End

This issue occurs when the Nios II application executes a BREAK instruction to transfer profiling data to the development workstation during the `exit()` or `return()` from `main()`.

As a workaround, call `usleep(500000)` before exiting or returning from `main()`. This call creates an adequate delay for you to transmit the I/O to the JTAG UART before `main` returns (or calls `exit()`). If the output is still partially truncated, increase the delay value passed to `usleep()`. Use `#include <unistd.h>` for the `usleep()` function prototype.

4.4.5.4. Fitting a Performance Counter in a Hardware Design That Consumes Most

During development, you can measure the system in a larger device than the size of your device in a deployed system.

Configure a performance counter to have only one section counter to save the most resources.

4.4.5.5. The Histogram for the gmon.out File Is Missing, Even Though My main()

If you do not define a system timer for the system, the profiler does not call the `nios2_pcsample()` function, and does not generate the histogram for the **gmon.out**. Define a system timer for your system.

4.5. Software System Design with a Nios II Processor Revision History

Document Version	Changes
2020.04.25	Corrected register information in section: *MPU Register Details*.
2020.01.13	Clarified Quartus support in *MPU Design Examples*
2017.06.12	Added a new section: *Profiling Systems*.
2016.12.19	Initial release.

Related Information

Document Revision History for Embedded Design Handbook on page 6

5. Nios II Configuration and Booting Solutions

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 Intel MAX 10, Intel FPGA Serial Flash (EPCS)/Intel FPGA Quad Serial Flash (EPCQ) configuration device, Quad Serial Parallel Interface (QSPI) flash or On-Chip RAM (OCRAM).

5.1. Introduction

The Nios II processor is a soft core processor that supports all Intel FPGA System on Chip (SoC) and Field Programmable Gate Array (FPGA) families. There are two types of supported boot processes—Execute-in-Place (XIP) and boot copier. Developing Nios II embedded programs can be based on the hardware abstraction layer (HAL). The HAL provides a small boot loader program (also known as boot copier) that copies relevant sections from the boot memory to their run time location at boot time. You can specify the run time locations for program and data memory by manipulating the Nios II BSP settings. The creation and management of software projects based on the HAL is integrated tightly with the Nios II Software Build Tools (SBT). The boot memory could be the Compact Flash Interface (CFI) flash, Intel MAX 10 User Flash Memory (UFM), Intel Serial Flash (EPCS)/Intel Quad Serial Flash (EPCQ) configuration device, Quad Serial Peripheral Interface (QSPI) flash or on-chip RAM (OCRAM). Regardless of the nature of the boot memory, HAL-based systems are constructed so that the reset vector and all program and data sections are initially stored in the boot memory.

This document describes:

- The Nios II processor boot copier that boots your Nios II system according to the boot memory selection
- Nios II processor booting options and general flow
- Nios II programming solutions for the selected boot memory

5.1.1. Prerequisites

You are required to have knowledge in instantiating and developing a system with a Nios II processor. Intel FPGA recommends that you go through the following online tutorials and training materials before using this application note:

- *Nios II Gen2 Software Developer's Handbook*
- *Nios II Flash Programmer User Guide*
- *AN370: Using the Serial Flash Loader with the Intel Quartus Prime Software*
- *Parallel Flash Loader IP Core User Guide*

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

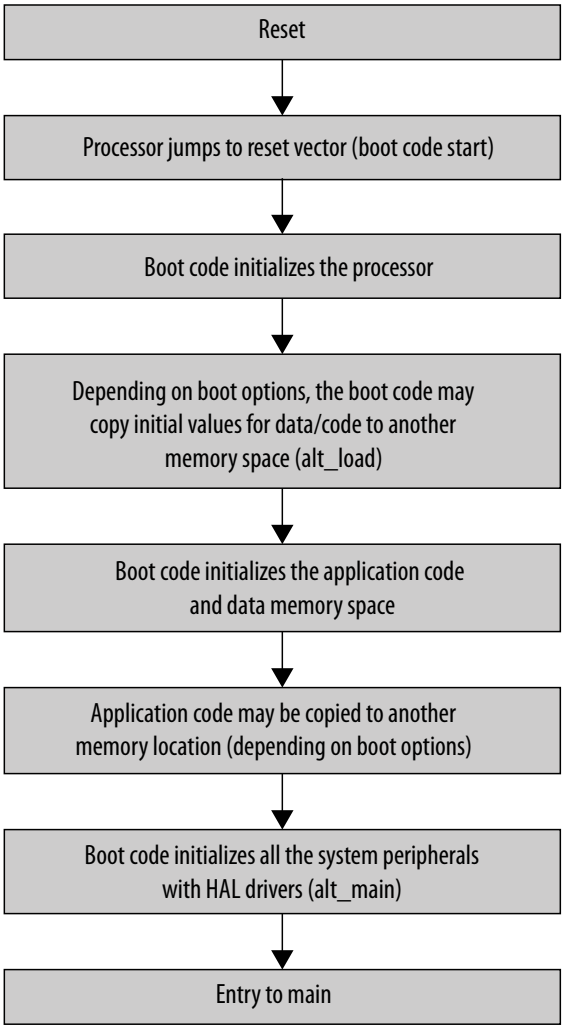
5.2. Nios II Processor Booting Methods

There are a few methods to boot up Nios II processor in Intel FPGA devices. The methods to boot up Nios II processor varies according to the flash memory selection.

Table 26. Supported Flash Memories with respective Boot Options

Supported Boot Memories	Nios II Booting Options	Application Code Storage	Application Runtime Location	Boot Copier
On-Chip Flash (UFM in Intel MAX 10 FPGA devices only)	Nios II processor application execute-in-place from On-chip Flash (UFM)	UFM	UFM (XIP) + OCRAM/ External RAM (for writable data sections)	Enable `alt_load()` function
	Nios II processor application copied from UFM to RAM using boot copier	UFM	OCRAM/ External RAM	Memcpy-based boot copier
EPCQ Flash	Nios II processor application execute-in-place from EPCQ flash	EPCQ flash	EPCQ flash (XIP) + OCRAM/ External RAM (for writable data sections)	Enable `alt_load()` function
	Nios II processor application copied from EPCQ flash to RAM using boot copier	EPCQ flash	OCRAM/ External RAM	Memcpy-based boot copier
QSPI Flash (applies to any FPGA with a soft QSPI Controller or Generic Serial Flash Interface Intel FPGA IP)	Nios II processor application execute-in-place from QSPI flash	QSPI flash	QSPI flash (XIP) + OCRAM/ External RAM (for writable data sections)	Enable `alt_load()` function
	Nios II processor application copied from QSPI flash to RAM using boot copier	QSPI flash	OCRAM/ External RAM	Memcpy-based boot copier
On-chip Memory (OCRAM)	Nios II processor application execute-in-place from OCRAM	OCRAM	OCRAM	No boot copier required
CFI Flash	Nios II processor application execute-in-place from CFI flash	CFI flash	CFI flash (XIP) + OCRAM/ External RAM (for writable data sections)	Enable `alt_load()` function
	Nios II processor application copied from CFI flash to RAM using boot copier	CFI flash	OCRAM/ External RAM	Memcpy-based boot copier

Figure 47. Nios II Boot Flow



5.2.1. Introduction to Nios II Booting Methods

Nios II systems require the hardware and software images to be configured in system memory before the processor can begin executing the application program. This section briefly describes various memory topologies that are available.

5.2.1.1. Nios II Processor Application Execute-In-Place from Boot Flash

The boot flash[5] controller is designed so that the boot flash address space is immediately accessible to the Nios II processor upon system reset, without the need to initialize the memory controller or memory devices. This enables Nios II to execute application code, stored on the boot devices, directly without the use of a boot copier to copy the code to another memory type.

[5] Boot Flash refers to UFM, EPCQ, QSPI, or CFI flash devices.

When a boot copier is not used, the Nios II application **.text** linker sections are set to the boot flash memory region while **.bss**, **.rodata**, **.rwdata**, **.stack** and **.heap** linker sections are set to the RAM memory region. Enable the `alt_load ()` function in the BSP Settings to copy the data sections (**.rodata**, **.rwdata**, **.exceptions**) to the RAM upon system reset. The code section (**.text**) remains in the boot flash memory region. (6)

5.2.1.1.1. `alt_load ()`

You may enable the `alt_load ()` function in the HAL code. This function operates as a mini boot copier that copies only the writable memory sections to RAM based on the BSP settings. It optionally copies sections from the boot memory to RAM. It can copy data sections (**.rodata**, **.rwdata**, **.exceptions**) to RAM but not the code sections (**.text**). The code section (**.text**) section is a read-only section and remains in the booting flash memory region. This partitioning helps to minimize the RAM usage but may limit the code execution performance because accesses to flash memory are slower than accesses to the on-chip RAM. The `alt_load ()` function can be enabled in the BSP Settings.

Table 27. BSP Settings

BSP Settings	Functions
`hal.linker.enable_alt_load`	Enable `alt_load()` function.
`hal.linker.enable_alt_load_copy_rodata`	`alt_load()` copies `.rodata` section to RAM.
`hal.linker.enable_alt_load_copy_rwdata`	`alt_load()` copies `.rwdata` section to RAM.
`hal.linker.enable_alt_load_copy_exceptions`	`alt_load()` copies `.exceptions` section to RAM.

5.2.1.2. Nios II Processor Application Copied from Boot Flash to RAM Using Boot Copier

The boot copier included with the Nios II processor, EDS, SBT, and HAL provides sufficient functionality for most Nios II applications and is convenient to implement with the Nios II software build tools development flow. If the system is not set up for OCRAM or XIP boot, then the Nios II processor boot copier is automatically and transparently added to your system software when you convert your executable files to flash programming files.

When a boot copier is used, all linker sections (**.text**, **.heap**, **.rwdata**, **.rodata**, **.bss**, **.stack**) are set to internal or external RAM. Using the boot copier to copy a Nios II processor application from boot flash(7) to internal or external RAM for execution helps to improve the execution performance. The Nios II SBT tool suite automatically adds the Nios II processor memcpy-based boot copier to the system when the executable file (**.elf**) is converted to the Memory initialization file (**.hex**) because memory access times are faster for RAM.

The boot copier is placed at the reset address, if the runtime location of the **.text** section is outside of the boot memory. Otherwise, the boot copier is located at the base address of the image or file, followed by the application, unless the **.text** linker

(6) **.text**, **.rodata**, **.rwdata**, and **.exceptions** are part of the Nios II default section names. For more details about these functions, refer to *Nios II Gen2 Software Developer's Handbook*.

(7) Boot Flash is referring to UFM, EPCS, EPCQ, CFI, or QPSI flash devices.

sections are set to the boot flash memory region. For this boot option, the Nios II processor starts executing the boot copier software upon system reset. The software copies the application from the boot flash to the internal or external RAM. Once this is complete, the Nios II processor transfers the program control over to the application.

The Nios II SBT ensures that the processor software is linked and determines where the software resides in memory. It uses the exception addresses to calculate where each code section is linked. The Nios II SBT positions the processor's code region in the memory component containing the exception address. Each processor has five default linker sections. The default linker sections are as follows:

- **.text**—the executable code
- **.rodata**—any read-only data used in the execution of the code
- **.rwdata**—where read-write variables and pointers are stored
- **.heap**—where dynamically allocated memory is located
- **.stack**—where function-call parameters and other temporary data is stored

SBT ensure that these sections are linked and located at fixed addresses in memory.

Note: If the boot copier is in flash, then the `alt_load ()` function does not need to be called because they both serve the same purpose.

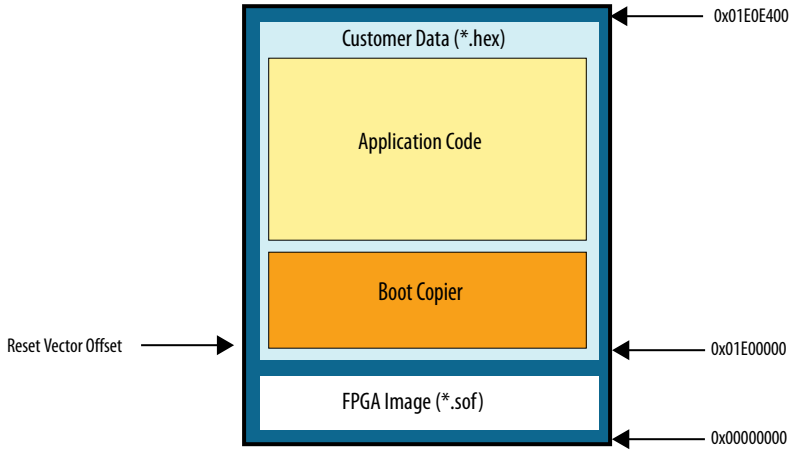
5.2.1.2.1. Memcpy-based Boot Copier

The memcpy-based boot copier is the default Nios II processor boot copier that supports Intel MAX 10 UFM, EPCS/ EPCQ, CFI, and QSPI flash memory. The memcpy-based boot copier has the following features:

- Locates the software application in non-volatile memory
- Unpacks and copies the software application image to Random Access Memory (RAM)
- Automatically switches processor execution to application code in RAM after copy completes

The boot image is located just right after the boot copier. You need to ensure the Nios II reset vector offset points to the start of the boot copier. Figure 48 on page 224 shows the flash memory map for EPCS/ EPCQ flash when using a boot copier. This memory map assumes a FPGA image is also stored in the flash memory.

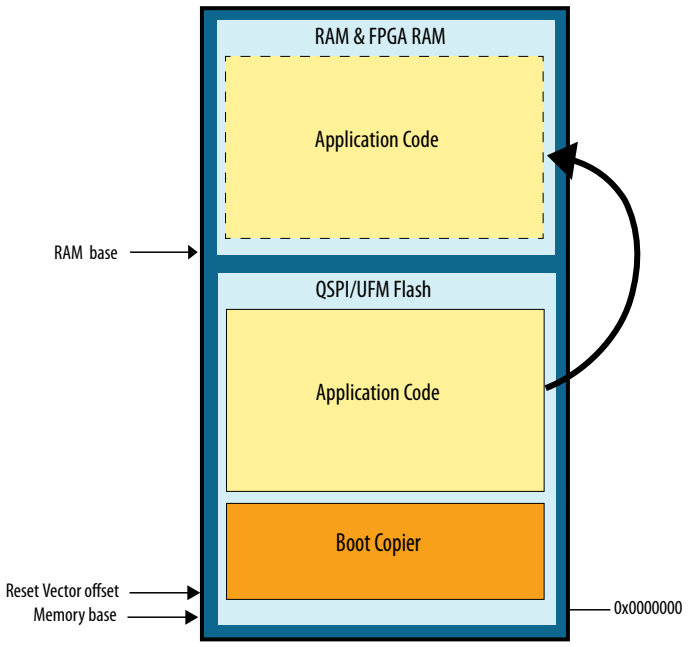
Figure 48. Memory Map for EPCS/EPCQ Flash with Memcpy-based Bootcopier



Note: At the start of the memory map is the FPGA image, followed by the customer data which consists of boot copier and application code. The size of the FPGA image is unknown and the exact size can only be known after the Intel Quartus Prime project compilation. The Nios II reset vector offset must be set in Platform Designer and must point to the start of the boot copier which is located after the FPGA image. You have to determine an upper bound for the size of the FPGA image. For instance, if the size of the FPGA image is estimated to be less than `0x01E00000`, you can set the Nios II Reset Vector offset to `0x01E00000` in Platform Designer, which is also the start of the boot copier.

Figure 49 on page 225 shows the memory map of a system designed for a Intel MAX 10 FPGA device using QSPI/UFM flash with the memcpy-based boot copier. Since the FPGA image (`*.sof`) is stored in the MAX10 CFM section, the boot copier is located at the base address of QSPI/UFM flash. Hence, the Nios II reset vector offset can be set to address `0x00000000` in Platform Designer.

Figure 49. Memory Map of a System Using UFM with memcpy-based Boot Copier



The memcpy-based boot copier is automatically incorporated as part of the HEX file during the memory initialization file generation ("`mem_init_generate`") target in the "Nios II Software Build Tools (SBT)" section.

For more information about `mem_init_generate`, refer to the "Nios II SBT Makefile "mem_init_generate" Target" section.

The source code for the memcpy-based boot copier is included with the Nios II Embedded Design Suite (EDS) under the following directory:

```
<install_directory>/<version>/nios2eds/components/altera_nios2/
boot_loader_sources
```

5.2.1.3. Nios II Processor Application Execute-In-Place from OCRAM

In this method, the Nios II reset address is set to the base address of the on-chip memory (OCRAM). The application (**.hex**) file is loaded into the OCRAM when the FPGA is configured, after the hardware design is compiled in the Intel Quartus Prime software. Once the Nios II processor resets, the application begins executing and branches to the entry point.

Note: This method does not require boot copier because Nios II application is already in place at system reset.

Note: The Nios II SBT tool suite does not include boot copier in the system if the **.text** Linker Section is set to OCRAM in the BSP Editor Settings.

5.2.2. Nios II Processor Booting from On-Chip Flash (UFM)

This section describes the Nios II processor booting and executing software from on-chip flash (UFM) available in Intel MAX 10 FPGA devices.

Intel MAX 10 FPGA devices contain on-chip flash that is segmented into two parts:

- Configuration Flash Memory (CFM)—stores the hardware configuration data for Intel MAX 10 FPGAs.
- User Flash Memory (UFM)—stores the user data or software applications.

The UFM architecture of Intel MAX 10 devices is a combination of soft and hard IPs. You can only access the UFM using the On-Chip Flash IP Core in the Intel Quartus Prime software.

5.2.2.1. Intel MAX 10 FPGA On-Chip Flash Description

The Intel MAX 10 On-chip Flash IP core supports the following features:

- Read or write accesses to UFM and CFM (if enabled in Platform Designer) sectors using the Avalon MM data and control slave interface.
- Supports page erase, sector erase and sector write.
- Simulation model for UFM read / write accesses using various EDA simulation tool.

Table 28. On-chip Flash Regions in Intel MAX 10 FPGA Devices

Flash Regions	Functionality
Configuration Flash Memory (sectors CFM0-2)	FPGA configuration file storage
User Flash Memory (sectors UFM0-1)	Nios II processor application and/or user data

Intel MAX 10 FPGA devices support several configuration modes and some of these modes allow CFM1 and CFM2 to be used as an additional UFM region. The following table shows the storage location of the FPGA configuration images based on the Intel MAX 10 FPGA's configuration modes.

Table 29. Storage Location of FPGA Configuration Images

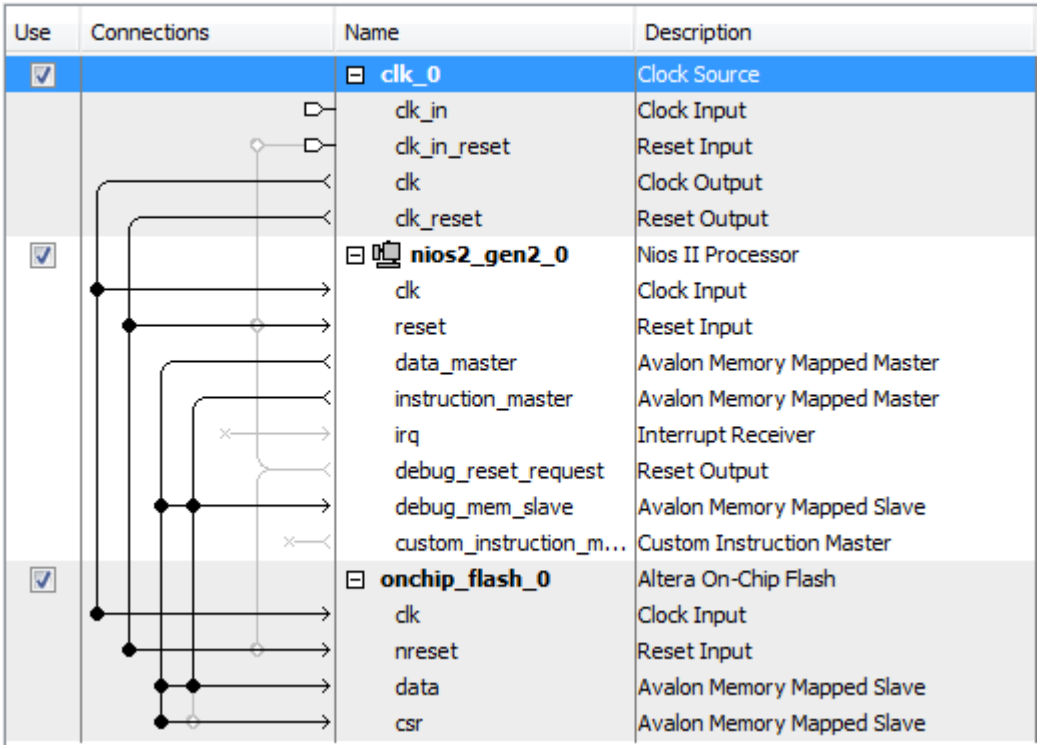
Configuration Mode	CFM2[8]	CFM1[8]	CFM0
Dual compressed images	Compressed Image 2		Compressed Image 1
Single uncompressed image	UFM[9]	Uncompressed image	
Single uncompressed image with Memory Initialization	Uncompressed image (with pre-initialized on-chip memory content)		
Single compressed image with Memory Initialization	Compressed image (with pre-initialized on-chip memory content)		
Single compressed image	UFM[9]		Compressed Image

(8) Sector is NOT supported in 10M02 device.

(9) The CFM sector is configured as virtual UFM.

You must use the Intel MAX 10 On-chip Flash IP core to access to the flash memory in Intel MAX 10 FPGAs. You can instantiate and connect the On-chip Flash IP to the Intel Quartus Prime software. The Nios II soft core processor uses the Avalon Memory-Mapped (Avalon-MM) interface to communicate with the On-chip Flash IP.

Figure 50. Connection Example for The On-chip Flash IP and the Nios II Processor



Note: Make sure the On-chip Flash `csr` port is connected to the Nios II processor `data_master` to enable the Nios II processor to control write and erase operations.

The On-chip Flash IP core can provide access to five flash sectors:

- UFM0
- UFM1
- CFM0
- CFM1
- CFM2

Important facts about UFM and CFM sectors:

- CFM sectors are intended for configuration (bitstream) data (*.pof) storage.
- User data can be stored in the UFM sectors and may be hidden, if the correct settings are selected in the Platform Designer tool.
- Certain devices do not have a UFM1 sector. You can refer to *Table: UFM and CFM Sector Size* for available sectors in each individual Intel MAX 10 FPGA device.

- You can configure CFM2 as a virtual UFM by selecting "Single Uncompressed Image" configuration mode.
- You can configure CFM2 and CFM1 as a virtual UFM by selecting "Single Compressed Image" configuration mode.
- The size of each sector varies with the selected Intel MAX 10 FPGA devices.

Table 30. UFM and CFM Sector Size

This table lists the dimensions of the UFM and CFM arrays.

Device	Pages per Sector					Page Size (Kbit)	Maximum User Flash Memory Size (Kbit)(10)	Total Configuration Memory Size (Kbit)	OCRAM Size (Kbit)
	UFM1	UFM0	CFM2	CFM1	CFM0				
10M02	3	3	0	0	34	16	96	544	108
10M04	0	8	41	29	70	16	1248	2240	189
10M08	8	8	41	29	70	16	1376	2240	378
10M16	4	4	38	28	66	32	2368	4224	549
10M25	4	4	52	40	92	32	3200	5888	675
10M40	4	4	48	36	84	64	5888	10752	1260
10M50	4	4	48	36	84	64	5888	10752	1638

Related Information

- MAX 10 FPGA Configuration User Guide
- MAX 10 User Flash Memory User Guide

5.2.2.2. Nios II Processor Application Execute-In-Place from UFM

This solution is suitable for Nios II processor applications which require limited on-chip memory usage. The `alt_load ()` function operates as a mini boot copier which copies the data sections (`.rodata`, `.rwdata`, or `.exceptions`) from boot memory to RAM based on the BSP settings. The code section (`.text`), which is a read only section, remains in the Intel MAX 10 On-chip Flash memory region. This setup minimizes the RAM usage but may limit the code execution performance as access to the flash memory is slower than the on-chip RAM.

The Nios II processor application is programmed into the UFM sector. The Nios II processor's reset vector points to the UFM base address in order to execute code from the UFM after the system resets.

If you are debugging the application using the source-level debugger, you must use a hardware break-point to debug because the UFM does not support random memory access. Random memory access is required for soft break-point debug.

(10) The maximum possible value, which is dependent on the configuration mode you select.

Figure 51. Nios II Application XIP from UFM

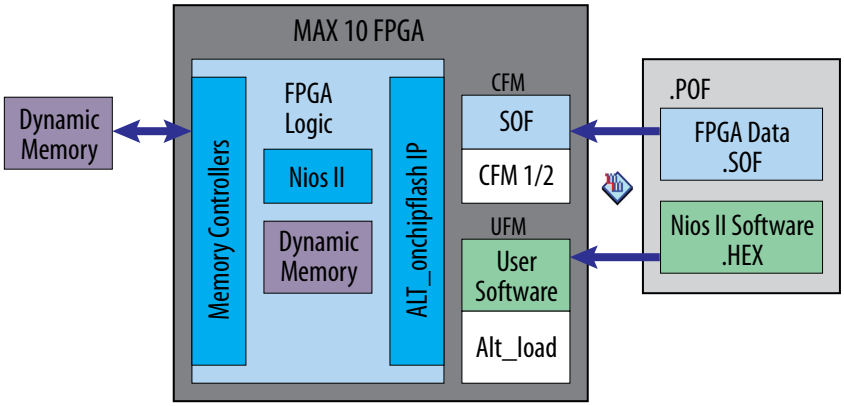
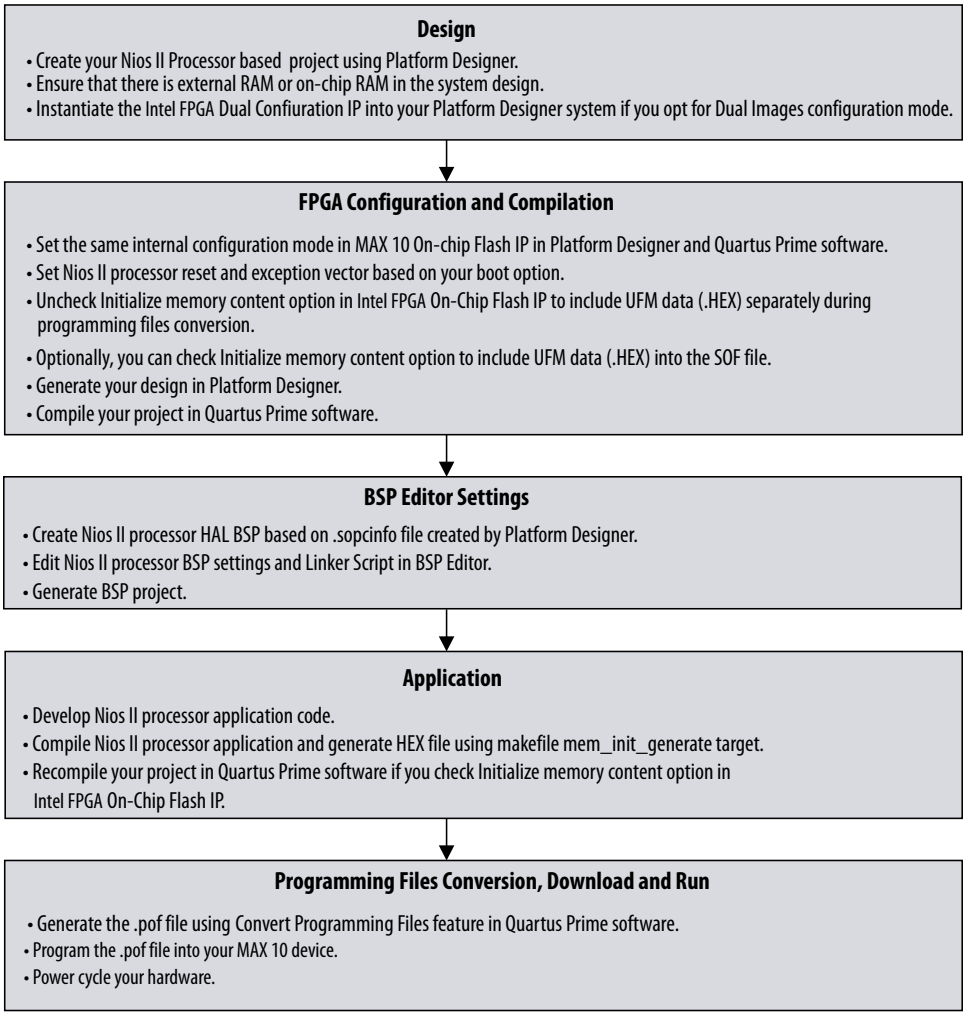


Table 31. RAM and ROM Size Requirement

You can manually determine the required RAM size for the On-Chip Flash by referring to the initial part of the `.objdump` file, created when you build your application.

RAM Size Requirement	ROM Size Requirement
Equivalent to the dynamic memory space usage during run time which is the sum of the maximum heap and stack size.	Executable code must not exceed the size of the UFM.

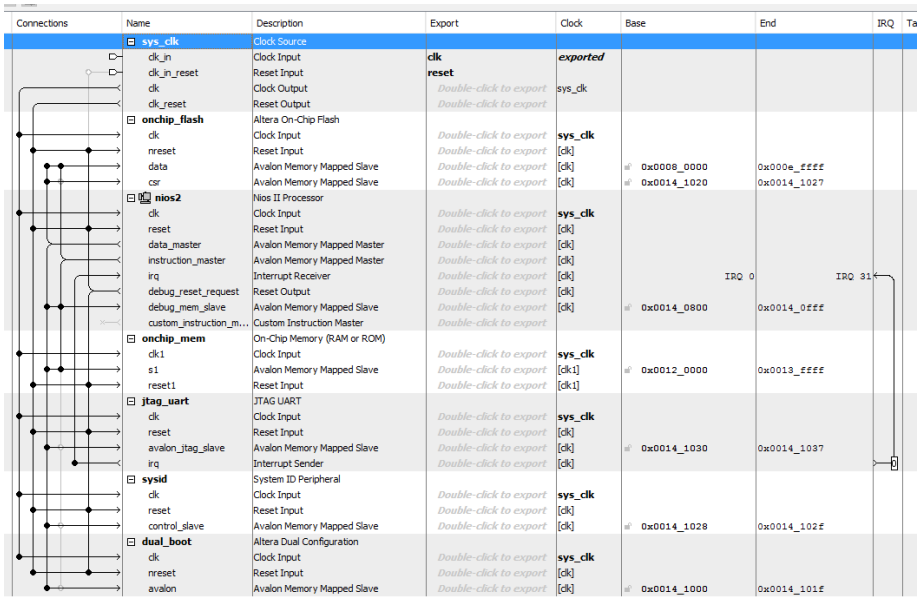
Figure 52. Design, Configuration and Booting Flow



5.2.2.2.1. Hardware Design

1. Create your Intel Quartus Prime and Platform Designer project.
2. Make sure external RAM or onchip memory (OCRAM) is added into your Platform Designer system.
3. If dual image configuration is required, ensure that the Dual Configuration IP is instantiated in Platform Designer (Configuration Mode is set to **Dual Compressed Images**).
4. Refer to the diagram below for example IP connections in Platform Designer.

Figure 53. Example IP Connections in Platform Designer for Booting Nios II from On-Chip Flash (UFM)



IP Component Settings

1. In the Nios II processor parameter editor, set the reset vector memory to On-Chip Flash and exception vector memory to OCRAM/External RAM or On-Chip Flash according to your design preference.

   Note: Setting the exception vector memory to OCRAM/External RAM is recommended to make the interrupt processing faster.

   a. If exception vector memory is set to On-Chip Flash, the minimum exception vector offset that you can set is 0x20.

      Note: When executing-in-place, the Nios II processor boots and runs directly from UFM, without copying any code at boot time. Because the Nios II begins executing at the reset address in this case, exception vectors must be located at a nonzero exception vector offset to allow for 32 bytes instructions between the reset vector and the base of the exception vectors.

Figure 54. Nios II Parameter Editor Settings with Exception Vector Set to OCRAM/ External RAM

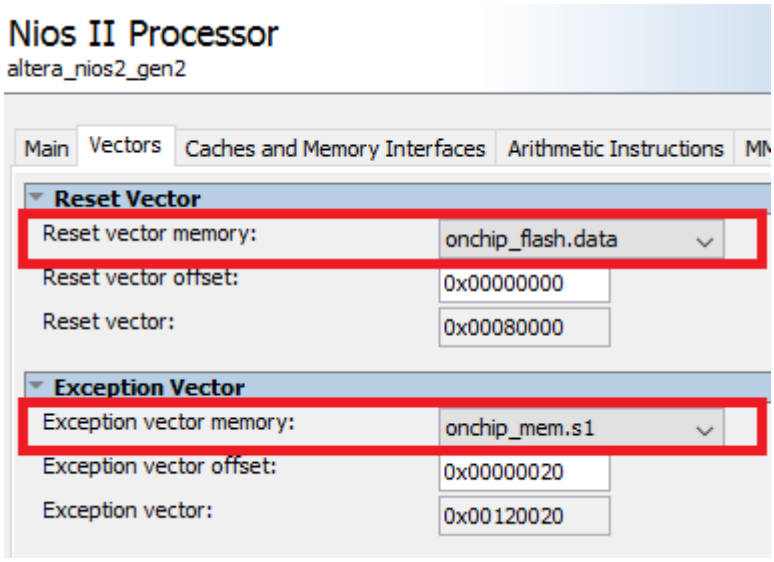
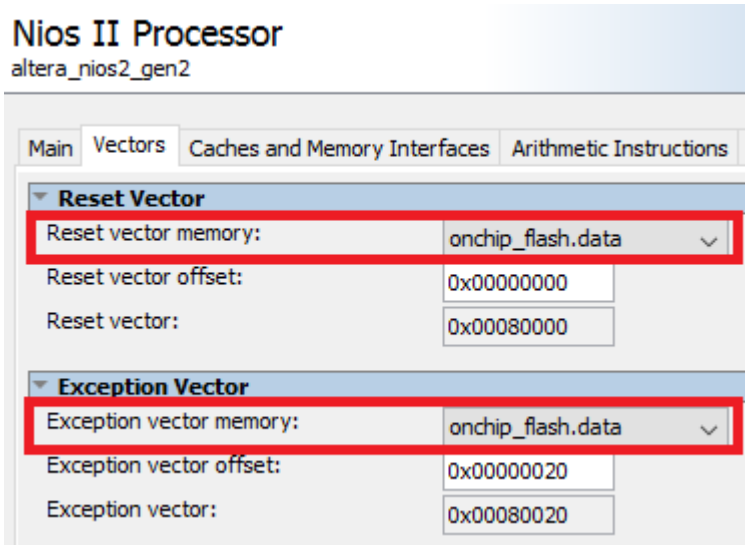
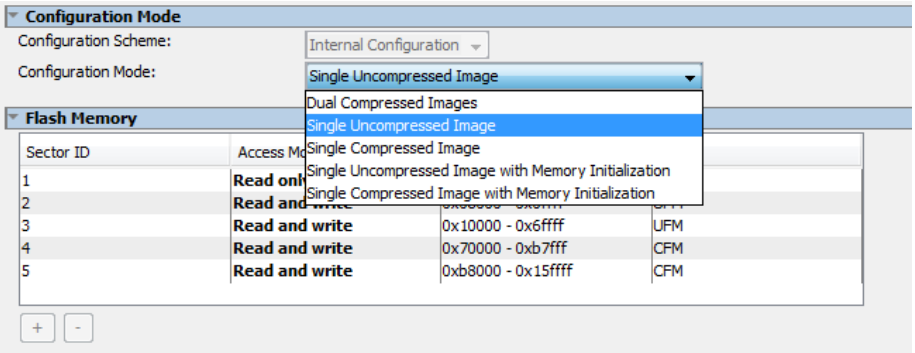


Figure 55. Nios II Parameter Editor Settings with Exception Vector Set to On-Chip Flash



2. In the On-chip Flash IP parameter editor, set the Configuration Mode to one of the following, according to your design preference:
   - Single Uncompressed Image
   - Single Compressed Image
   - Dual Compressed Images

Figure 56. Configuration Mode Selection in On-Chip Flash Parameter Editor



3. Program the UFM data (HEX file) by using one of the following methods:
   - **Method 1: Initialize the UFM data in the SOF**—Intel Quartus Prime includes the UFM initialization data in the SOF during compilation. SOF recompilation is needed if there are changes in the UFM data.
   - **Method 2: Combine UFM data with a compiled SOF during programming files (POF) generation**(11)—UFM data is combined with the compiled SOF during the programming files conversion. SOF recompilation is NOT needed even if there are changes in the UFM data.
4. Click **Generate HDL**. The **Generation** dialog box appears.
5. Specify output file generation options, and then click **Generate**.

On-Chip Flash IP Settings for Method 1

1. Check **Initialize flash content**.

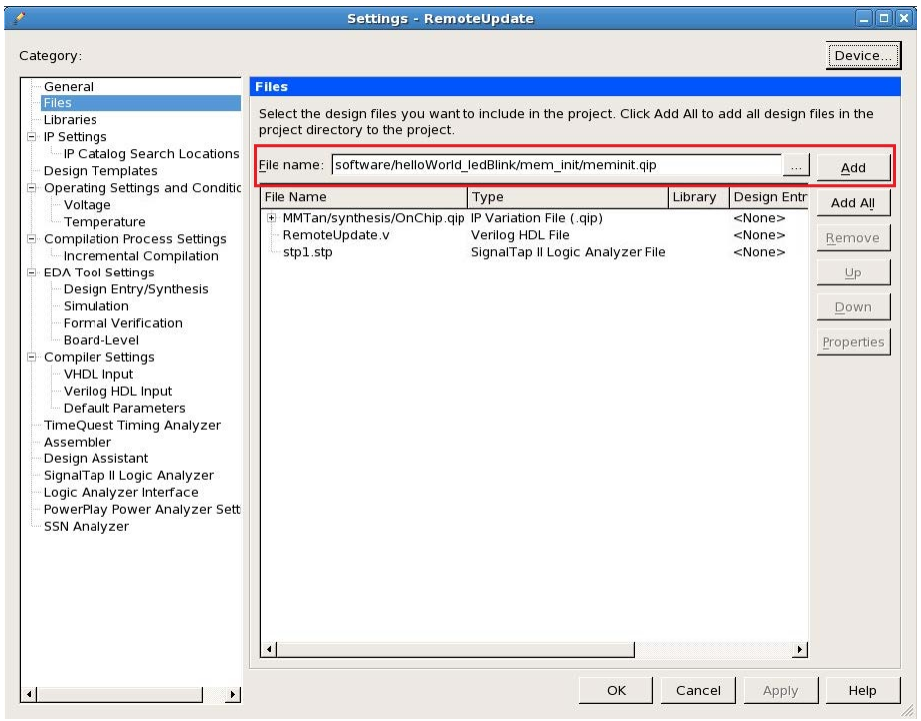
Figure 57. Initialize Flash Contents with Default Initialization File



2. If the default path is used, add `meminit.qip` generated during "make mem_init_generate" into Intel Quartus Prime project.

(11) This is the recommended method for application developer. You are not required to recompile SOF file for application changes during development.

Figure 58. Adding `meminit.qip` File in Intel Quartus Prime



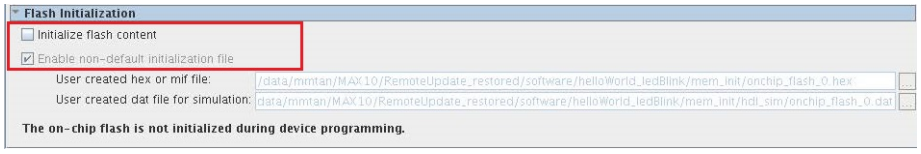
Make sure the generated HEX naming matches the default naming.

3. If non-default path is selected, enable the **Enable non-default initialization file** and specify the path of the HEX file.

On-Chip Flash IP Settings for Method 2

1. Uncheck **Initialize flash content**.

Figure 59. Initialize Flash Content with Non-default Initialization File

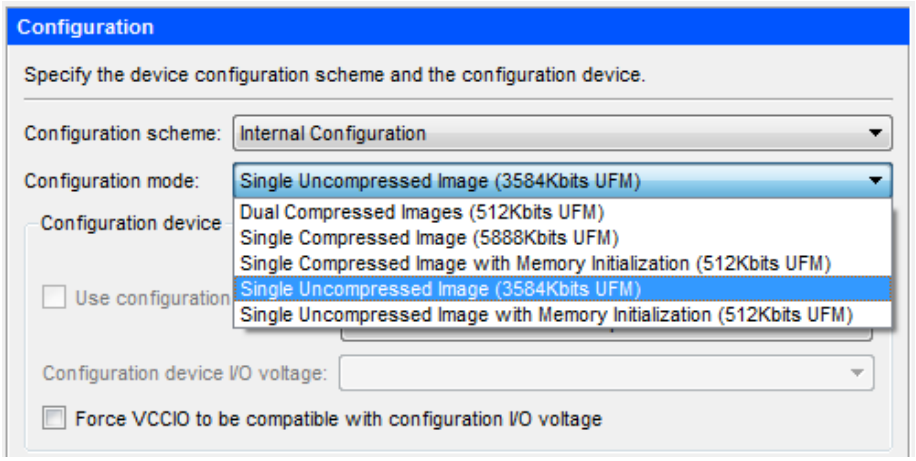


Intel Quartus Prime Software Settings

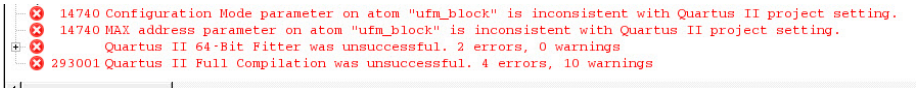
1. In the Intel Quartus Prime software, click on **Assignment ➤ Device ➤ Device and Pin Options ➤ Configuration**. Set **Configuration mode** to **Single Uncompressed Image** or **Single Compressed Image** or **Dual Compressed Images**.(12)

(12) The size of UFM shown vary according to your device selection.

Figure 60. Configuration Mode Selection in Intel Quartus Prime Software



Note: If the configuration mode setting in Intel Quartus Prime software and Platform Designer parameter editor is different, the Intel Quartus Prime project fails with the following error messages.



2. Click **OK** to exit the **Device and Pin Options window**.
3. Click **OK** to exit the **Device** window.
4. Click **Processing > Start Compilation** to compile your project and generate the `.sof` file.

5.2.2.2.2. Application

In the Nios II SBT window, select either:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.

  or

- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

BSP Editor Settings

You must edit the BSP settings according to the selected Nios II processor boot options.

1. In the Nios II SBT tool, right click on your BSP project in the **Project Explorer** window. Select **Nios II ➤ BSP Editor...** to open the **Nios II BSP Editor**.
2. In Nios II BSP Editor, expand into **Advanced.hal.linker** under **Settings** in **Main** tab.
3. If the exception vector memory is set to OCRAM or External RAM, then enable the following:

- **allow_code_at_reset**
- **enable_alt_load**
- **enable_alt_load_copy_rodata**
- **enable_alt_load_copy_rwdata**
- **enable_alt_load_copy_exceptions**

Figure 61. Advanced.hal.linker Settings

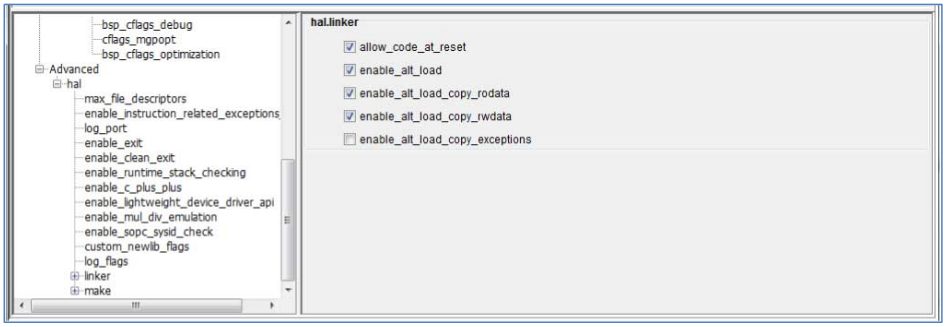
If exception vector memory is set to OCRAM or External RAM.



4. If the exception vector memory is set to On-chip Flash, then enable the following:
   - **allow_code_at_reset**
   - **enable_alt_load**
   - **enable_alt_load_copy_rodata**
   - **enable_alt_load_copy_rwdata**

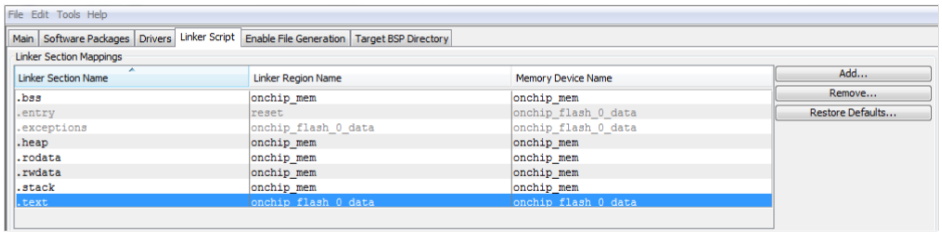
Figure 62. Advanced.hal.linker Settings

If exception vector memory is set to On-Chip Flash.



5. Click on the **Linker Script** tab in the Nios II BSP Editor.
6. Set the **.text** region in the **Linker Section Name** to the On-chip Flash in the **Linker Region Name**. Set the rest of the regions in the **Linker Section Name** list to the On-chip Memory (OCRAM) or external RAM.

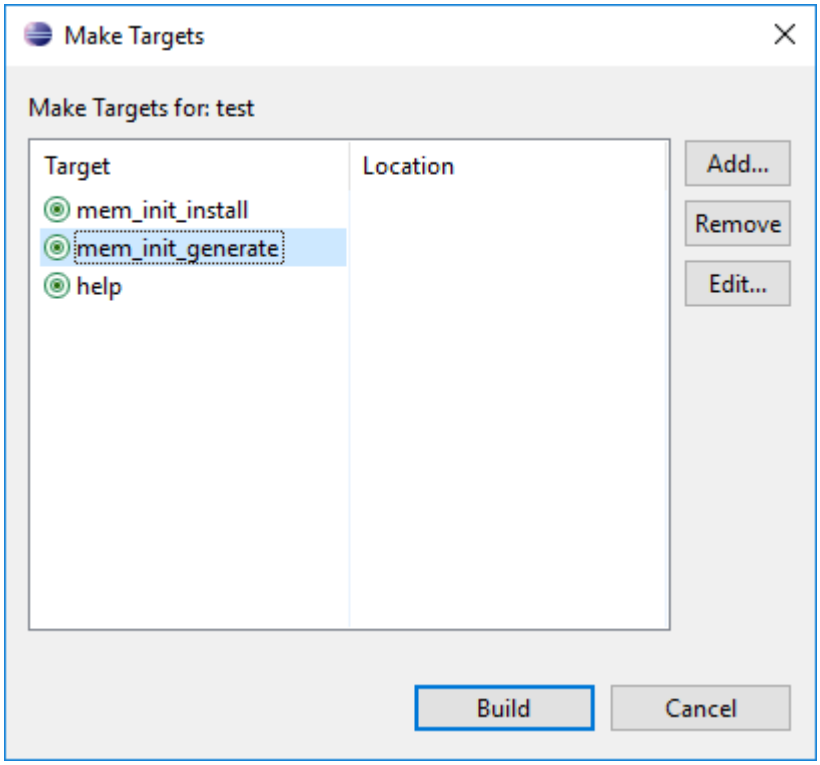
Figure 63. Linker Region Settings



HEX File Generation

1. After creating the Nios II application, right click on the project in **Project Explorer** and select **Build Project**. An ***.elf** file is created under the project folder.
2. Again right click on your project and click **Make Targets** -> **Build...**, the **Make Targets** dialog box appears. You can also press shift + F9 to trigger the **Make Target** dialog box.
3. Select **mem_init_generate**.
4. Click **Build** to generate the HEX file.

Figure 64. Selecting `mem_init_generate` in Make Targets



5. The "mem_init_generate" macro will create two HEX files; `<on_chip_ram.hex>` and `<on_chip_flash.hex>`.

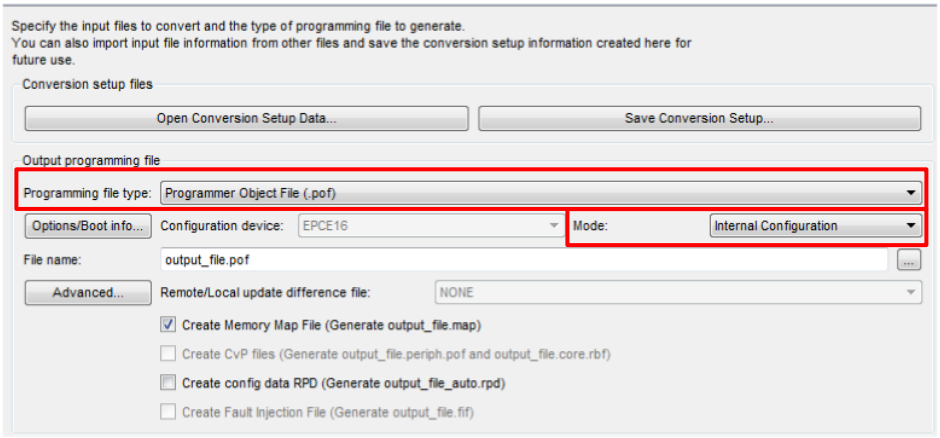
Note:
- The `mem_init_generate` target also generates a Intel Quartus Prime IP file (`meminit.qip`). Intel Quartus Prime software will refer to the `meminit.qip` for the location of the initialization files.
- All these files can be found under the "`<project_folder>/software/<application_name>/mem_init`" folder.

6. Recompile your project in Intel Quartus Prime software if you check **Initialize memory content** option in On-Chip Flash IP. This is to include the software data (.HEX) into the SOF file.

5.2.2.2.3. Programming

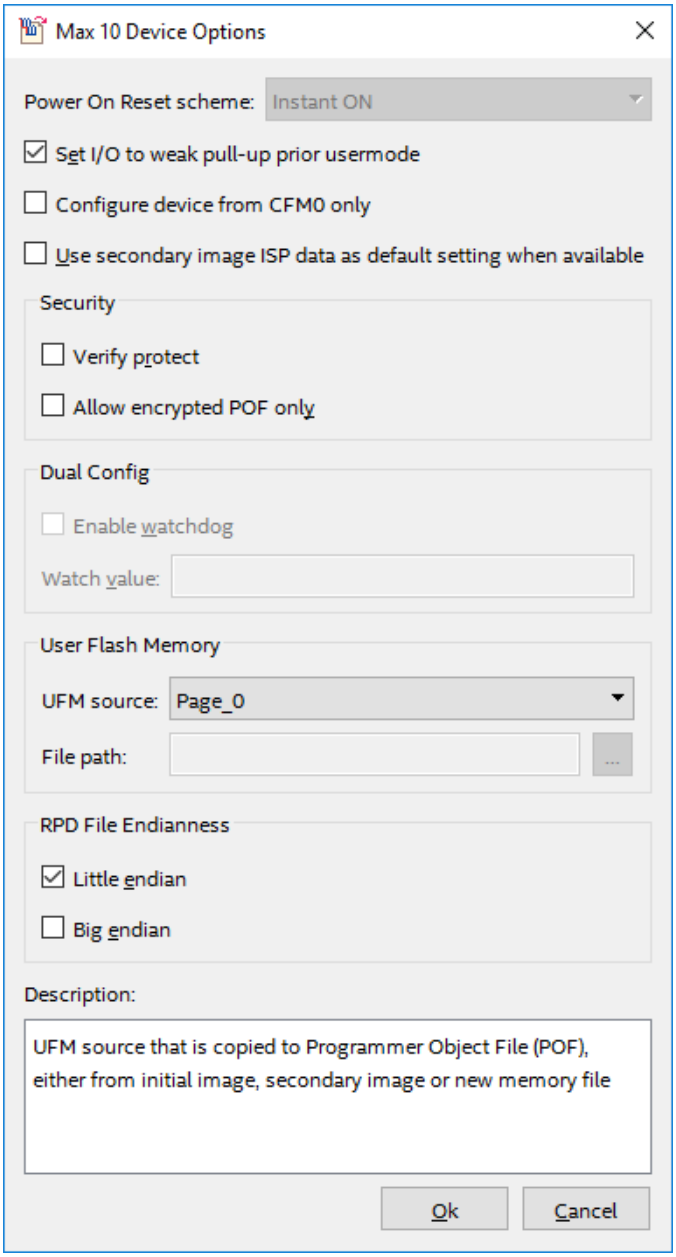
1. In Intel Quartus Prime, click on **Convert Programming Files** from the **File** tab.
2. Choose **Programmer Object File (.pof)** as **Programming file type:**.
3. Set **Mode** to **Internal Configuration**.

Figure 65. Convert Programming File Settings



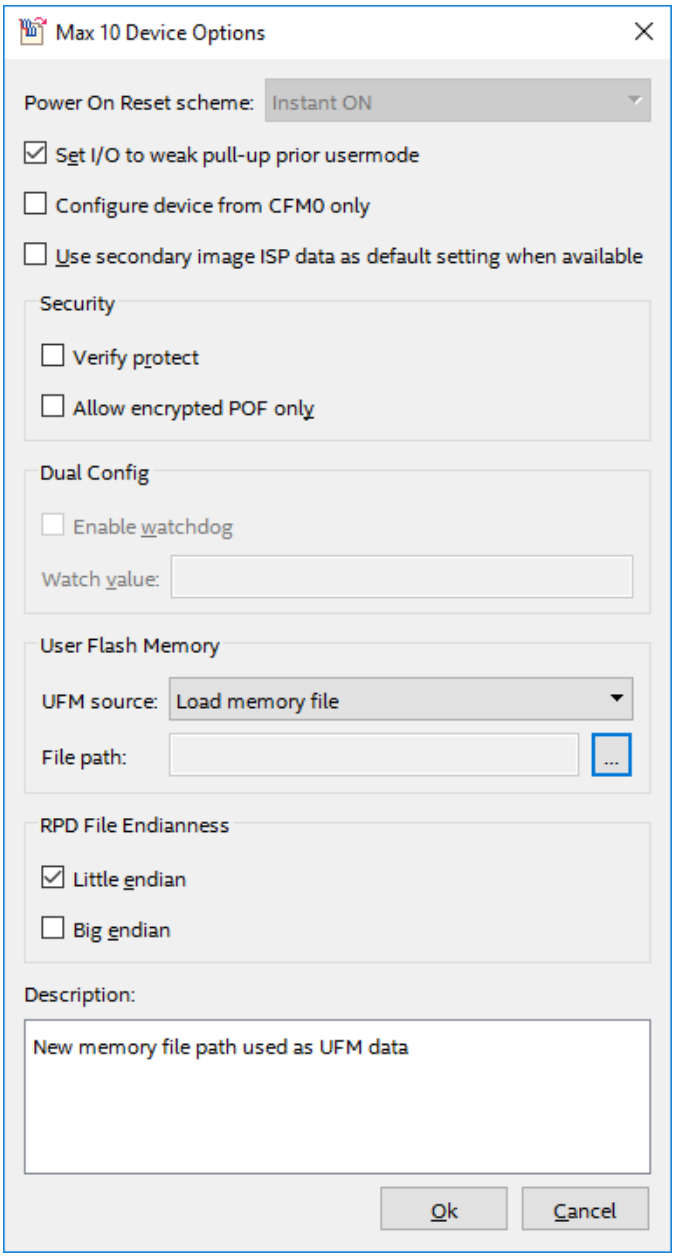
4. Click on **Options/Boot info...**, the Intel MAX 10 Device Options dialog box appears.
5. Based on the **Initialize flash content** settings in the On-chip Flash IP, do one of the following:
   - If **Initialize flash content** is checked, the UFM initialization data was included in the SOF during Intel Quartus Prime compilation.
     - For **Single Uncompressed/ Compressed Image** configuration mode, select **Page_0** for **UFM source:** option. Click **OK** and proceed to next step.
     - For **Dual Compressed Images** configuration mode, select **Page_0** or **Page_1** for **UFM source:** option. Click **OK** and proceed to next step. The **UFM data** (.HEX file) can be included in either **Page_0** or **Page_1** only. The On-chip flash does not support two .HEX files for Dual Compressed images configuration mode.

Figure 66. Setting Page_0 for UFM Source if Initialize Flash Content is checked



— If **Initialize flash content** is not checked, choose **Load memory file** for the **UFM source** option. Browse to the generated On-chip Flash HEX file (`on_chip_flash.hex`) in the **File path:** and click **OK**. This will add UFM data separately to the SOF file during the programming file conversion.

Figure 67. Setting Load Memory File for UFM Source if Initialize Flash Content is not checked



6. In the Convert Programming File dialog box, at the **Input files to convert** section, click **Add File...** and point to the generated Intel Quartus Prime .sof file.

Figure 68. Input Files to Convert in Convert Programming Files for Single Image Mode

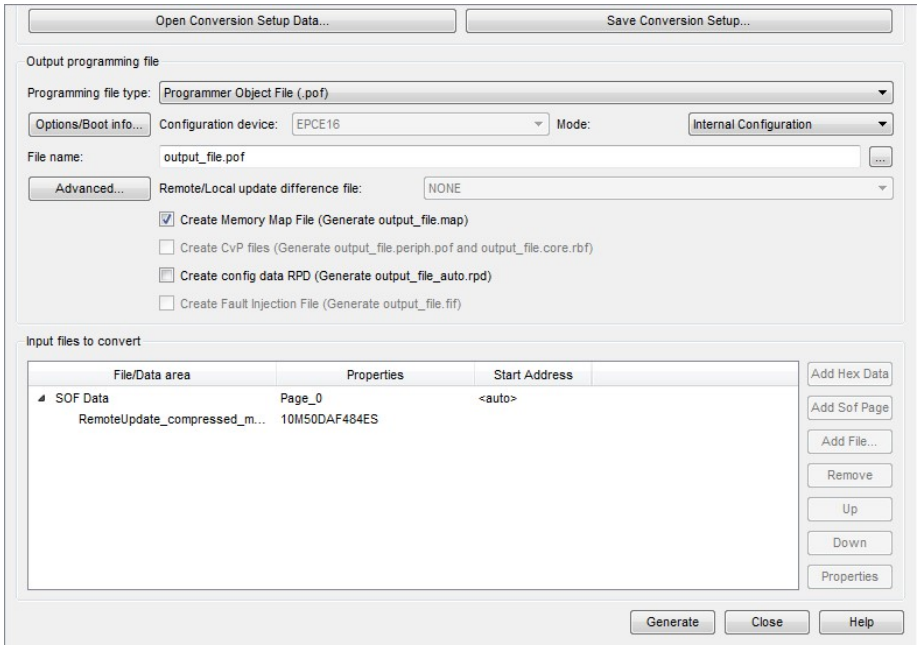
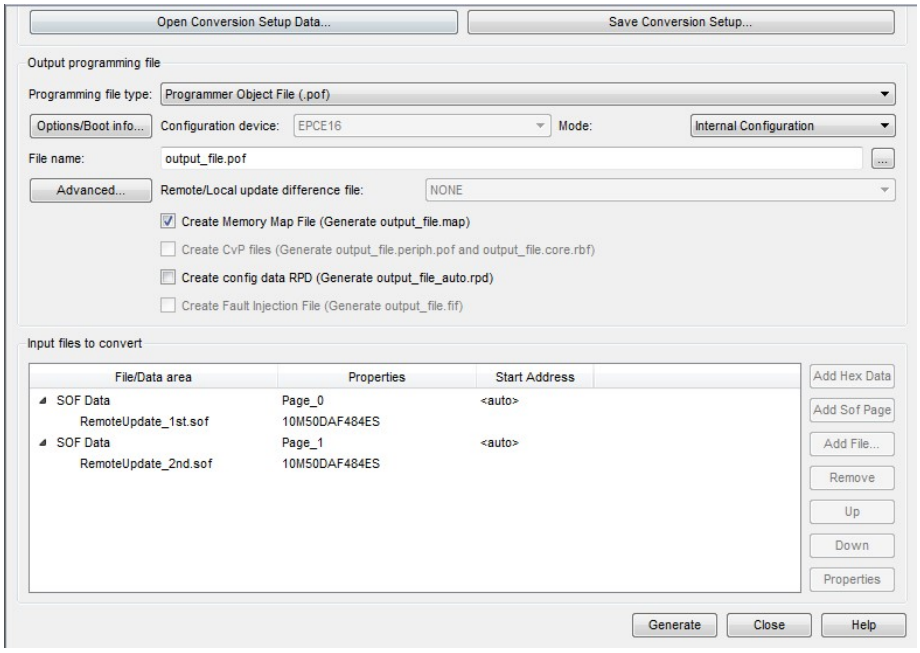


Figure 69. Input Files to Convert in Convert Programming Files for Dual Images Mode



7. Click **Generate** to create the .pof file.
8. Program the .pof file into your Intel MAX 10 device.
9. Power cycle your hardware.

5.2.2.3. Nios II Processor Application Copied from UFM to RAM using Boot Copier

Intel recommends this solution for Intel MAX 10 FPGA Nios II processor system designs where there may be multiple iterations of application software development and when high system performance is required. The boot copier is located within the UFM at an offset which is the same address with the reset vector. The Nios II application is located next to the boot copier.

For this boot option, the Nios II processor starts executing the boot copier upon system reset to copy the application from the UFM sector to the OCRAM or external RAM. Once copying is complete, the Nios II processor transfers the program control over to the application.

Figure 70. Nios II Application Copied from UFM to RAM using Boot Copier

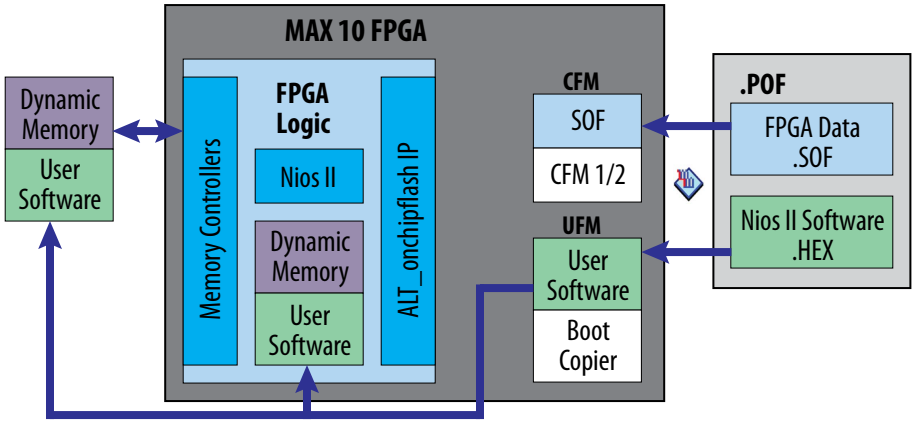
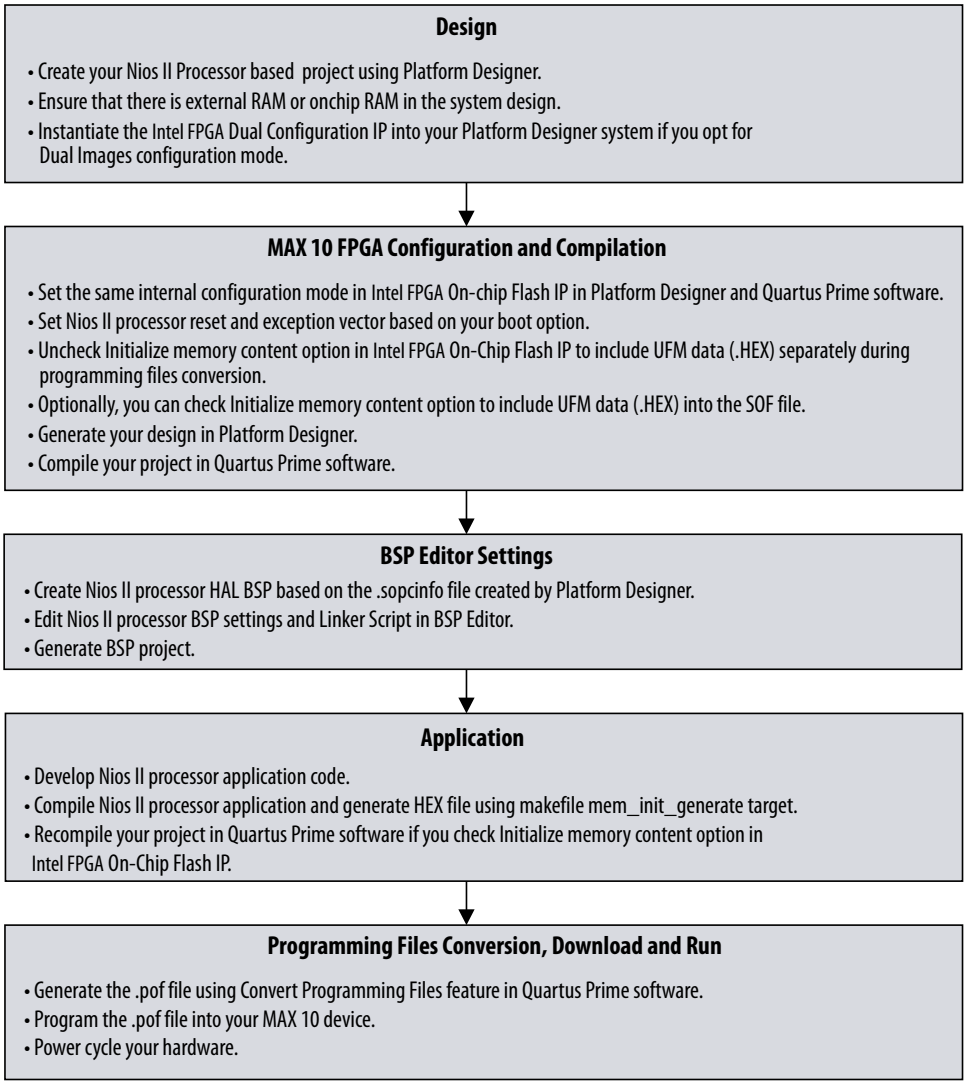


Table 32. RAM and ROM Size Requirement

- You can manually determine the required RAM size for the On-Chip Flash by referring to the initial part of the `.objdump` file, created when you build your application.
- The OCRAM size is limited, you have to ensure that the size is sufficient for application execution.

RAM Size Requirement	ROM Size Requirement
Equivalent to the executable code and dynamic memory size required by user program.	Executable code and boot copier must not exceed the size of the UFM.

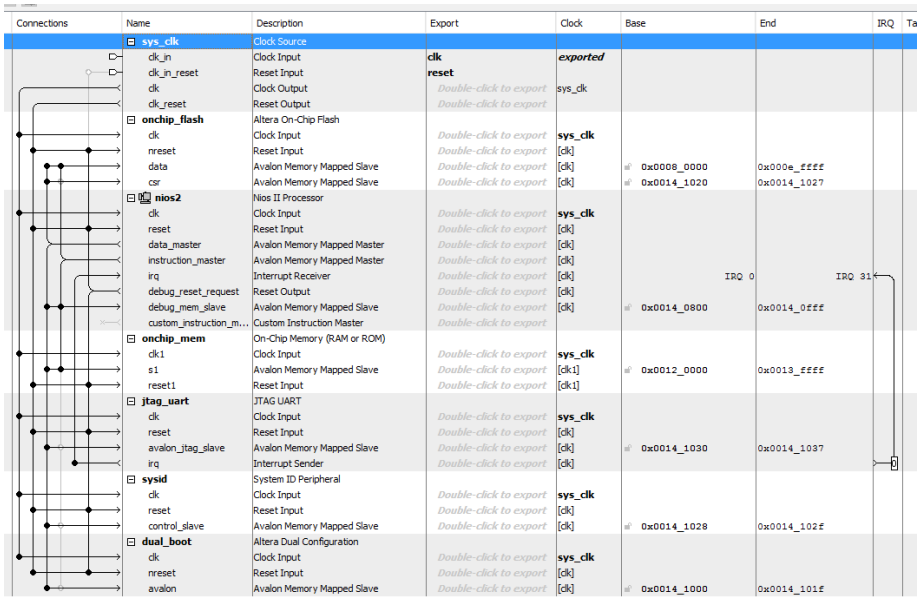
Figure 71. Configuration and Booting Flow



5.2.2.3.1. Hardware Design

1. Create your Intel Quartus Prime and Platform Designer project.
2. Make sure external RAM or onchip memory (OCRAM) is added into your Platform Designer system.
3. If dual image configuration is required, ensure that the Dual Configuration IP is instantiated in Platform Designer (Configuration Mode is set to **Dual Compressed Images**).
4. Refer to the diagram below for example IP connections in Platform Designer.

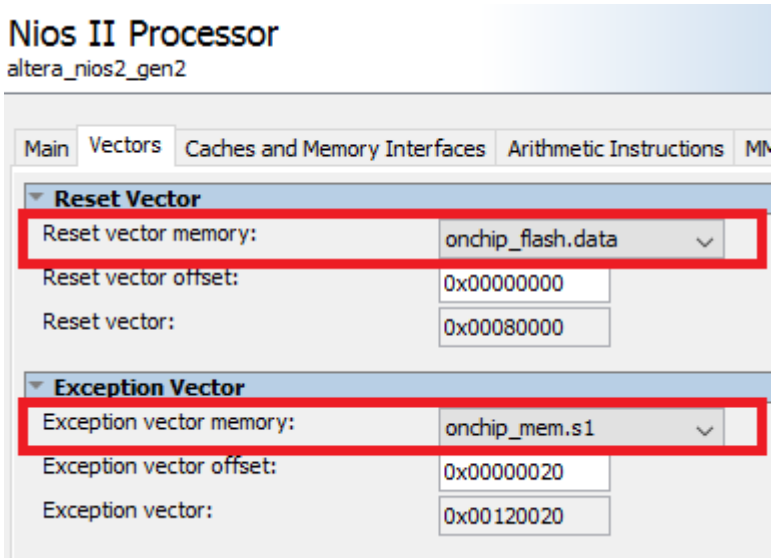
Figure 72. Example IP Connections in Platform Designer for Booting Nios II from On-Chip Flash (UFM)



IP Component Settings

1. In the Nios II processor parameter editor, set the reset vector memory to On-Chip Flash and exception vector memory to OCRAM/ External RAM.

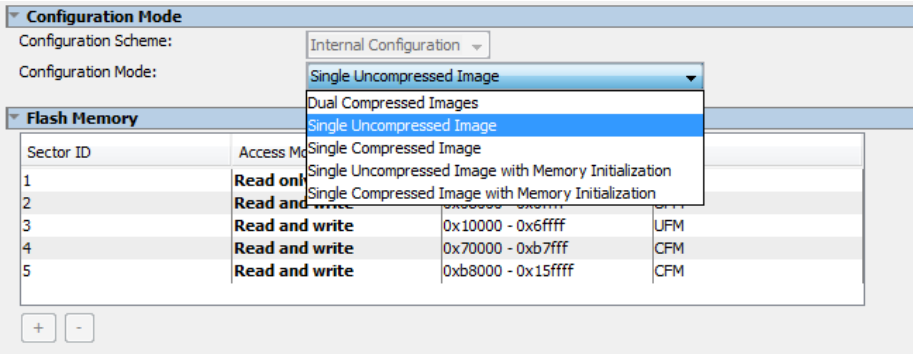
Figure 73. Nios II Parameter Editor Settings



2. In the On-chip Flash IP parameter editor, set the Configuration Mode to one of the following, according to your design preference:

- Single Uncompressed Image
- Single Compressed Image
- Dual Compressed Images

Figure 74. Configuration Mode Selection in On-Chip Flash Parameter Editor



3. Program the UFM data (HEX file) by using one of the following methods:
   - **Method 1: Initialize the UFM data in the SOF**—Intel Quartus Prime includes the UFM initialization data in the SOF during compilation. SOF recompilation is needed if there are changes in the UFM data.
   - **Method 2: Combine UFM data with a compiled SOF during programming files (POF) generation**(13)—UFM data is combined with the compiled SOF during the programming files conversion. SOF recompilation is NOT needed even if there are changes in the UFM data. .
4. Click **Generate HDL**. The **Generation** dialog box appears.
5. Specify output file generation options, and then click **Generate**.

On-Chip Flash IP Settings for Method 1

1. Check **Initialize flash content**.

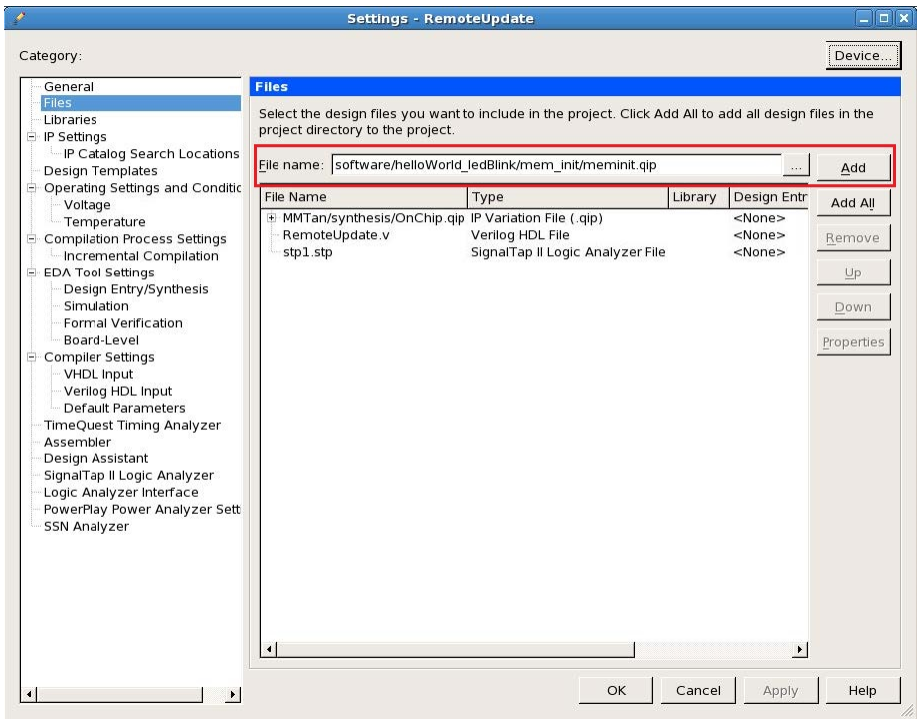
Figure 75. Initialize Flash Contents with Default Initialization File



2. If the default path is used, add `meminit.qip` generated during "make mem_init_generate" into Intel Quartus Prime project.

(13) This is the recommended method for application developer. You are not required to recompile SOF file for application changes during development.

Figure 76. Adding `meminit.qip` File in Intel Quartus Prime



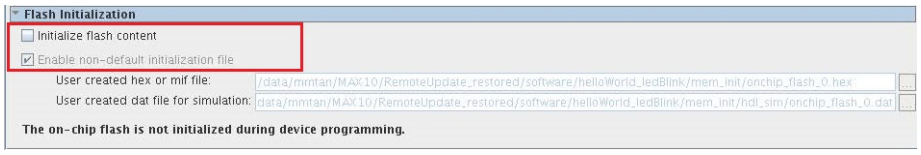
Make sure the generated HEX naming matches the default naming.

3. If non-default path is selected, enable the **Enable non-default initialization file** and specify the path of the HEX file.

On-Chip Flash IP Settings for Method 2

1. Uncheck **Initialize flash content**.

Figure 77. Initialize Flash Content with Non-default Initialization File

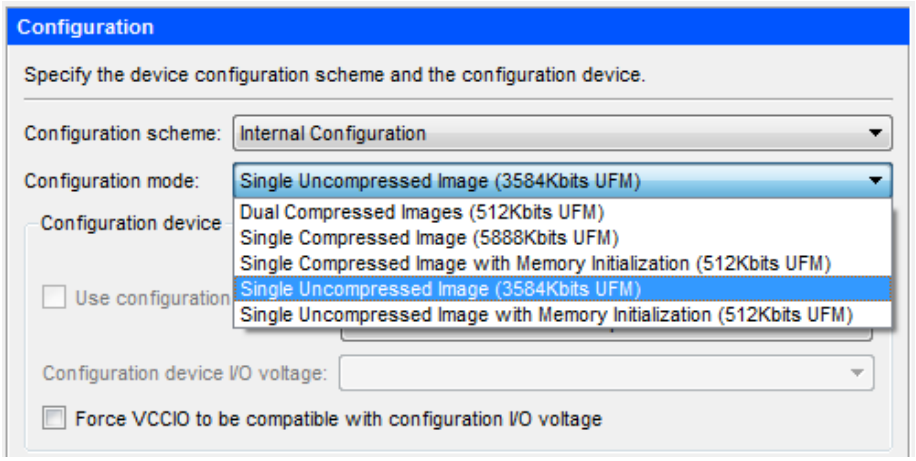


Intel Quartus Prime Software Settings

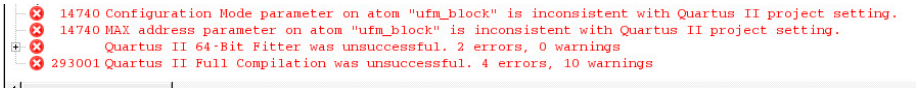
1. In the Intel Quartus Prime software, click on **Assignment ➤ Device ➤ Device and Pin Options ➤ Configuration**. Set **Configuration mode** to **Single Uncompressed Image** or **Single Compressed Image**.(14)

(14) The size of UFM shown vary according to your device selection.

Figure 78. Configuration Mode Selection in Intel Quartus Prime Software



Note: If the configuration mode setting in Intel Quartus Prime software and Platform Designer parameter editor is different, the Intel Quartus Prime project compilation will fail with the following error message.



2. Click **OK** to exit the **Device and Pin Options window**.
3. Click **OK** to exit the **Device** window.
4. Click **Processing > Start Compilation** to compile your project and generate the `.sof` file.

5.2.2.3.2. Application

In the Nios II SBT window, select either:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.

  or

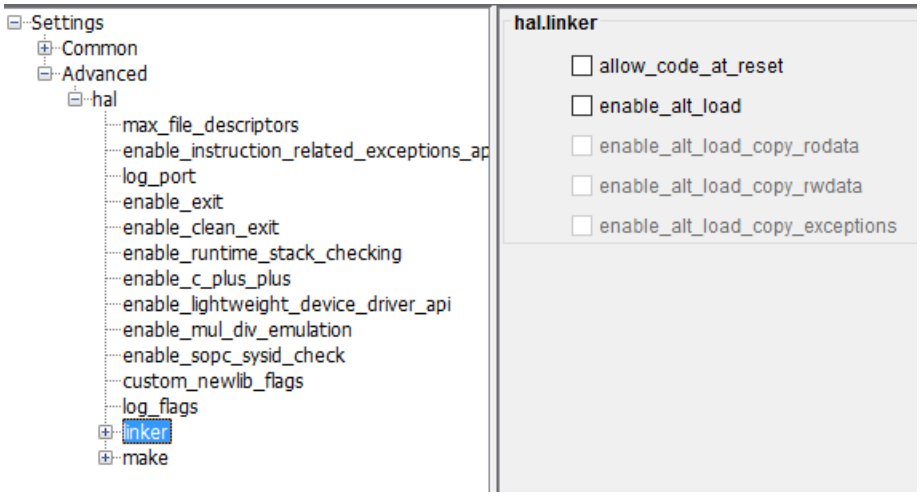
- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

BSP Editor Settings

You must edit the BSP editor settings according to the selected Nios II processor boot options.

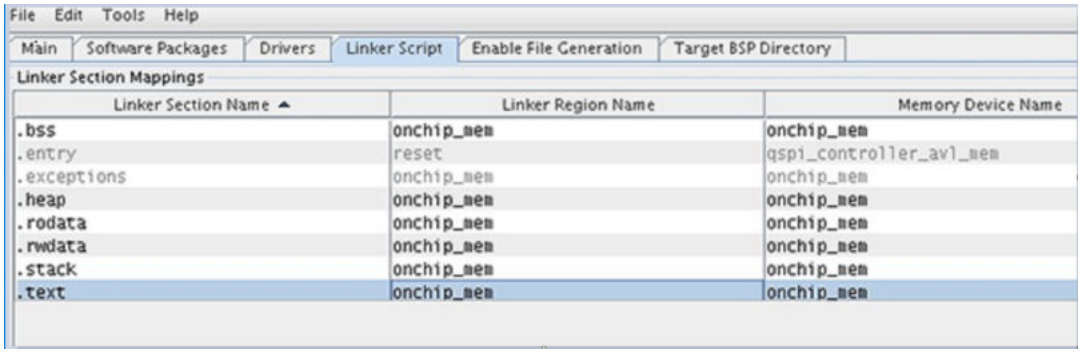
1. In the Nios II SBT tool, right click on your BSP project in the **Project Explorer** window. Select **Nios II ➤ BSP Editor...** to open the **Nios II BSP Editor**.
2. In Nios II BSP Editor, expand into **Advanced.hal.linker** under **Settings** in **Main** tab.
3. Leave all settings unchecked.

Figure 79. Advanced.hal.linker Settings



4. Click on the **Linker Script** tab in the Nios II BSP Editor.
5. Set all of the items in the **Linker Section Name** list to the On-chip Memory (OCRAM) or external RAM.

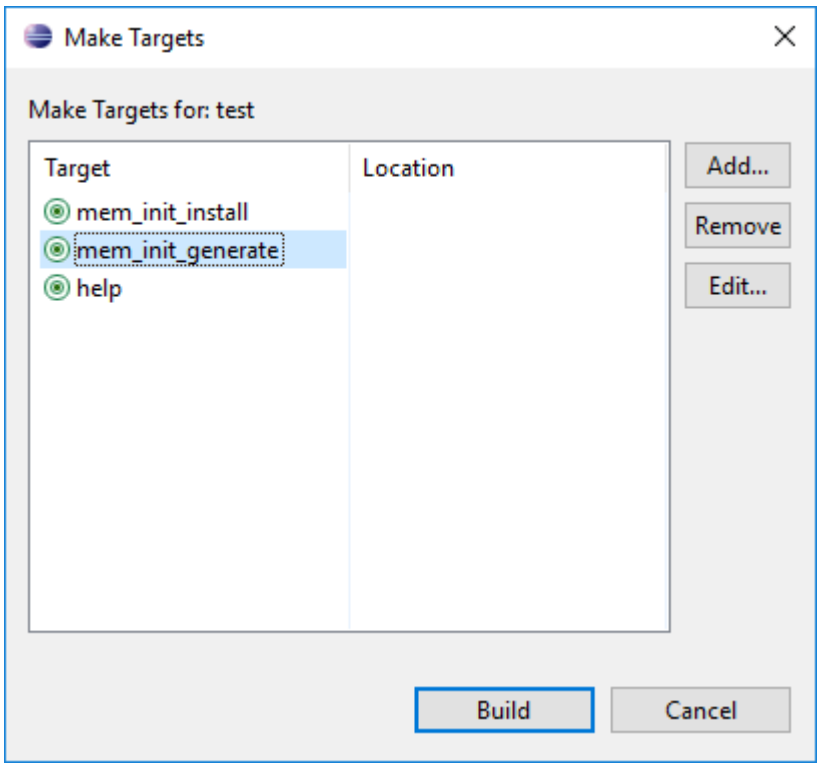
Figure 80. Linker Region Settings



HEX File Generation

1. After creating the Nios II application, right click on the project in **Project Explorer** and select **Build Project**. An ***.elf** file is created under the project folder.
2. Again right click on your project and click **Make Targets** -> **Build...**, the **Make Targets** dialog box appears. You can also press shift + F9 to trigger the **Make Target** dialog box.
3. Select **mem_init_generate**.
4. Click **Build** to generate the HEX file.

Figure 81. Selecting `mem_init_generate` in Make Targets



5. The "mem_init_generate" macro will create two HEX files; `<on_chip_ram.hex>` and `<on_chip_flash.hex>`.

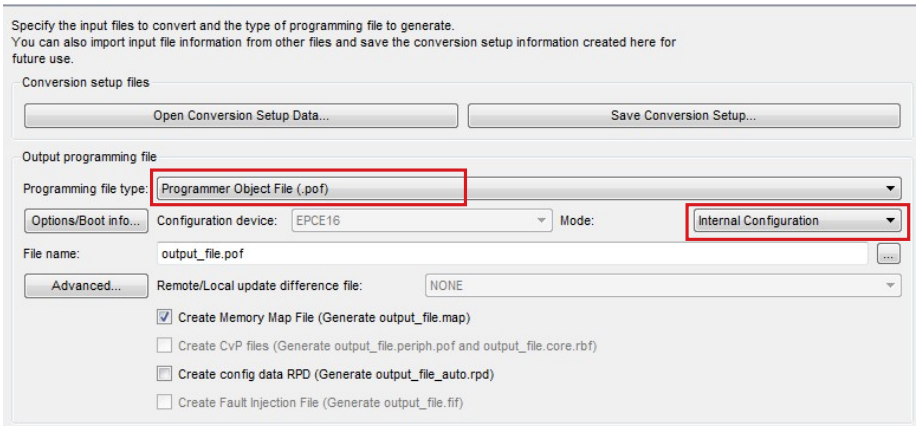
   Note:
   - The `mem_init_generate` target also generates a Intel Quartus Prime IP file (`meminit.qip`). Intel Quartus Prime software will refer to the `meminit.qip` for the location of the initialization files.
   - All these files can be found under the "`<project_folder>/software/<application_name>/mem_init`" folder.

6. Recompile your project in Intel Quartus Prime software if you check **Initialize memory content** option in On-Chip Flash IP. This is to include the software data (.HEX) into the SOF file.

5.2.2.3.3. Programming

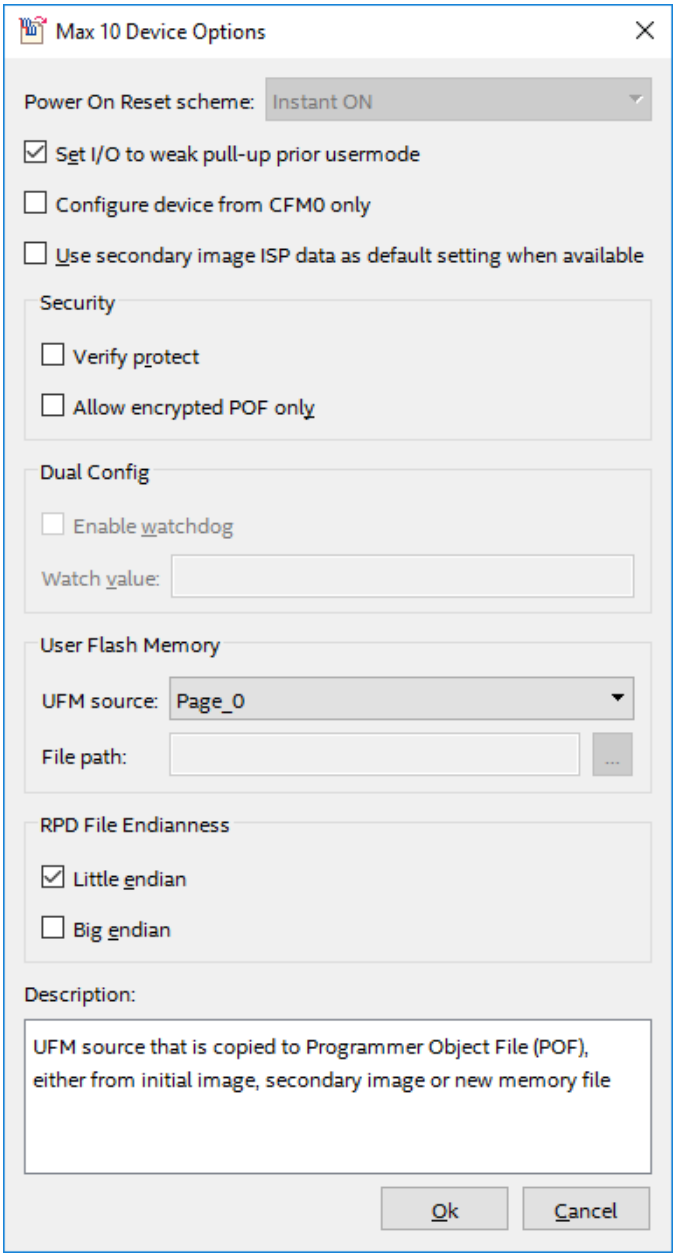
1. In Intel Quartus Prime, click on **Convert Programming Files** from the **File** tab.
2. Choose **Programmer Object File (.pof)** as **Programming file type:**.
3. Set **Mode** to **Internal Configuration**.

Figure 82. Convert Programming File Settings



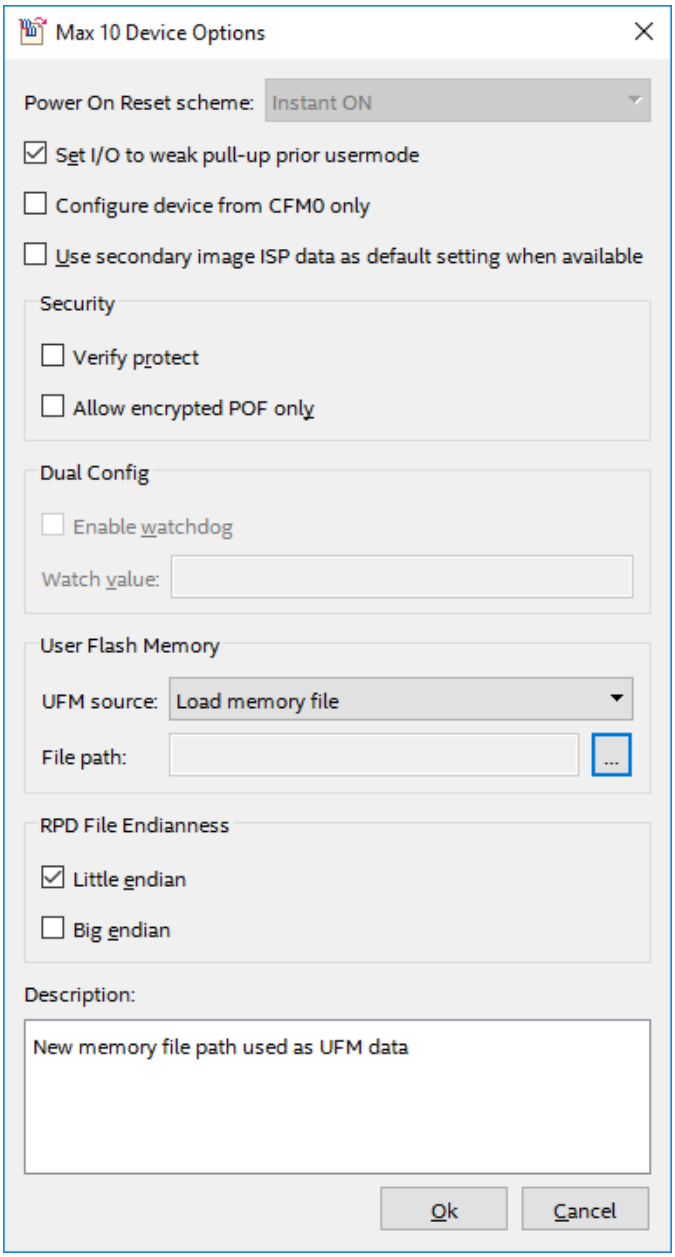
4. Click on **Options/Boot info...**, the Intel MAX 10 Device Options dialog box appears.
5. Based on the **Initialize flash content** settings in the On-chip Flash IP, do one of the following:
   - If **Initialize flash content** is checked, the UFM initialization data was included in the SOF during Intel Quartus Prime compilation.
     - For **Single Uncompressed/ Compressed Image** configuration mode, select **Page_0** for **UFM source:** option. Click **OK** and proceed to next step.
     - For **Dual Compressed Images** configuration mode, select **Page_0** or **Page_1** for **UFM source:** option. Click **OK** and proceed to next step. The **UFM data** (.HEX file) can be included in either **Page_0** or **Page_1** only. The On-chip flash does not support two .HEX files for Dual Compressed images configuration mode.

Figure 83. Setting Page_0 for UFM Source if Initialize Flash Content is checked



— If **Initialize flash content** is not checked, choose **Load memory file** for the **UFM source** option. Browse to the generated On-chip Flash HEX file (`on_chip_flash.hex`) in the **File path:** and click **OK**. This will add UFM data separately to the SOF file during the programming file conversion.

Figure 84. Setting Load Memory File for UFM Source if Initialize Flash Content is not checked



6. In the Convert Programming File dialog box, at the **Input files to convert** section, click **Add File...** and point to the generated Intel Quartus Prime .sof file.

Figure 85. Input Files to Convert in Convert Programming Files for Single Image Mode

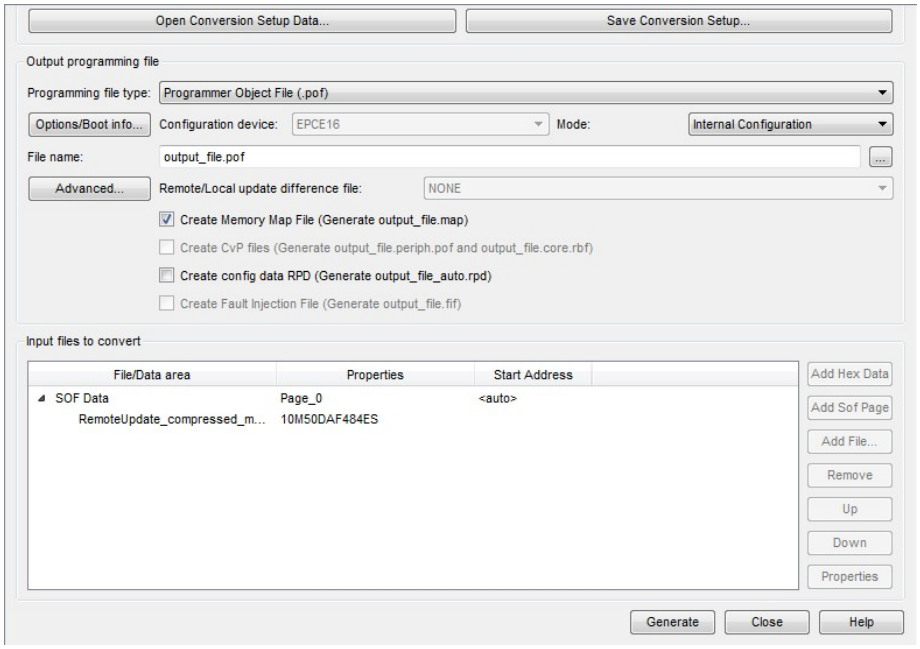
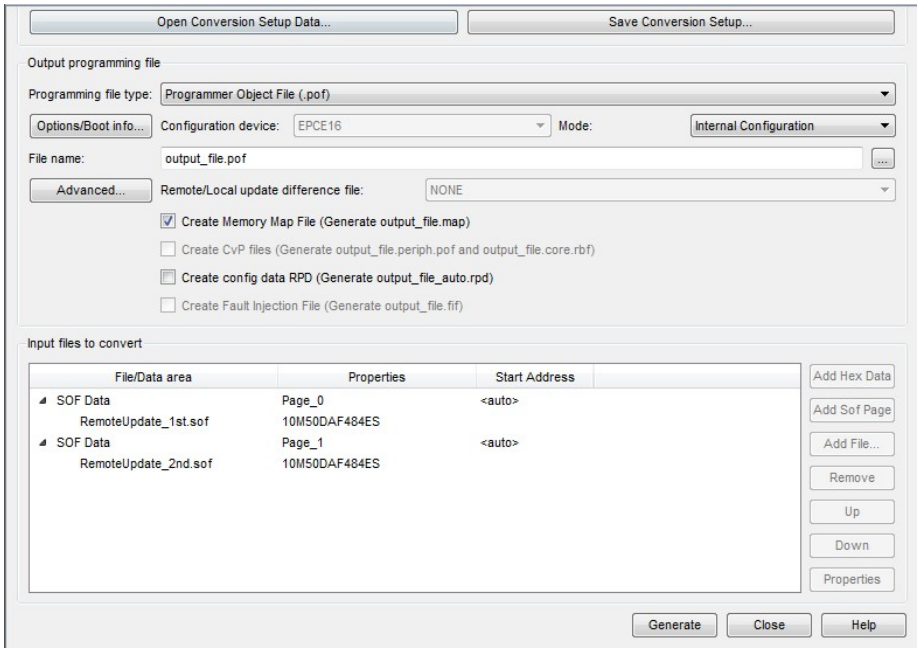


Figure 86. Input Files to Convert in Convert Programming Files for Dual Images Mode



7. Click **Generate** to create the .pof file.
8. Program the .pof file into your Intel MAX 10 device.
9. Power cycle your hardware.

5.2.3. Nios II Processor Booting from EPCQ Flash

This section describes how to boot a Nios II processor from EPCQ flash memory (EPCQx1, EPCQx4) using an Intel FPGA Serial Flash Controller. The Intel FPGA Serial Flash Controller supports configuring and programming of EPCQx4, EPCQx1 and EPCS flash in FPGA design for Intel FPGA devices.

5.2.3.1. Intel FPGA Serial Flash Controller (EPCQ) Overview

The Intel FPGA Serial Flash Controller allows Nios II processor systems to access EPCS/ EPCQ flash memories, which support standard or quad I/O mode. The Nios II processor SBT supports the Nios II booting from the Intel FPGA Serial Flash Controller. In addition, a Nios II hardware abstraction layer (HAL) driver is available for the Intel FPGA Serial Flash Controller that allows an application to read, write, or erase flash.

The Nios II processor supports the following two boot options using EPCQ flash:

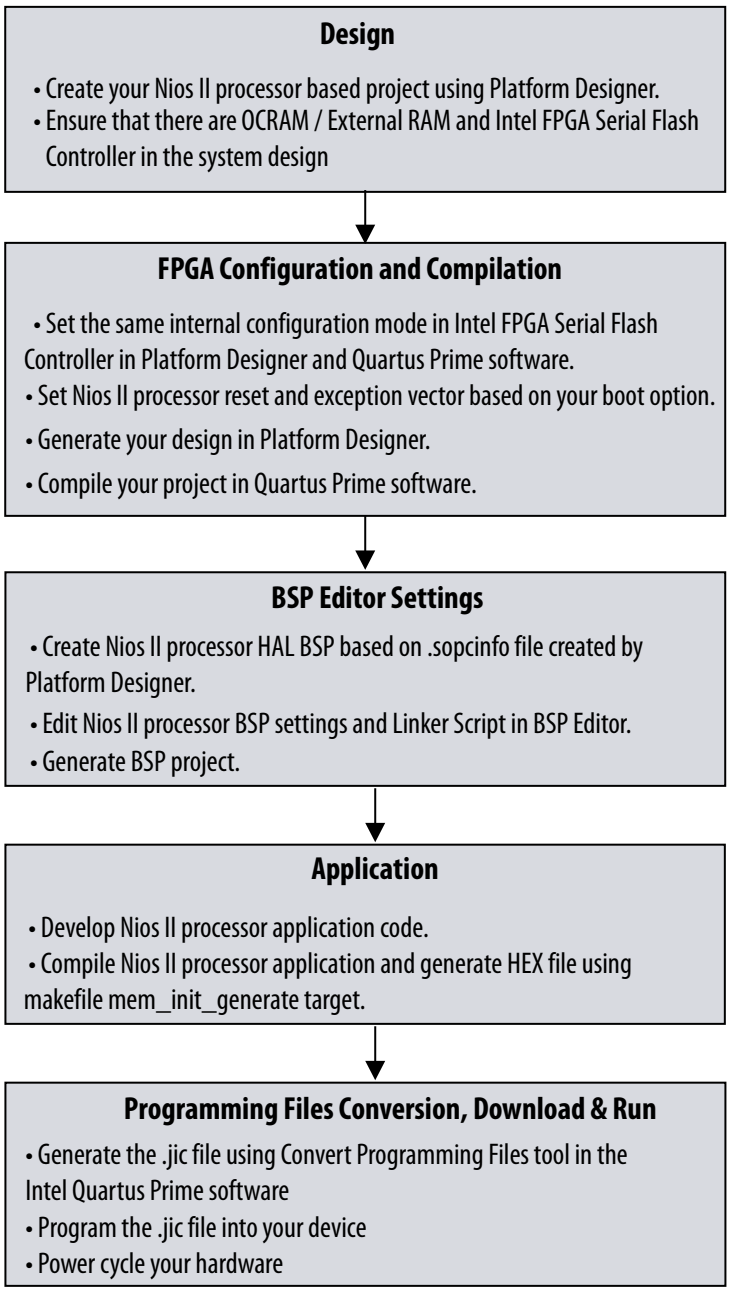
- Nios II processor application executes in place from EPCQ flash
- Nios II processor application is copied from EPCQ flash to RAM using a boot copier

Table 33. Summary of Nios II Processor Boot Options Using EPCQ Flash

Boot Option	Application Code Stored Location	Application Run-time Location	Method
Nios II processor application execute-in-place from EPCQ flash	EPCQ flash	EPCQ flash (XIP) + OCRAM/ External RAM (for writable data sections)	Using `alt_load ()` function
Nios II processor application copied from EPCQ flash to RAM using boot copier	EPCQ flash	OCRAM/ External RAM	Using Memcpy-based boot copier

5.2.3.2. Nios II Processor Design, Configuration, and Boot Flow

Figure 87. Design, Configuration, and Boot Flow



5.2.3.3. Nios II Processor Application Execute-In-Place from EPCQ Flash

The execute-in-place option is suitable for Nios II processor applications which require limited on-chip memory usage.

The `alt_load ()` function operates as a mini boot copier which initializes and copies only the writable memory sections to on-chip RAM (OCRAM). The code section (**.text**), which is a read-only section, remains in the EPCQ flash memory region. Retaining the read-only section in EPCQ helps to minimize RAM use, but may limit the code execution performance. The Nios II processor application is programmed into the EPCQ flash.

The Nios II processor reset vector points to the EPCQ flash to allow code execution after the system resets. If you are debugging the application using the source-level debugger, you must use a hardware breakpoint because the EPCQ cannot efficiently support random memory access.

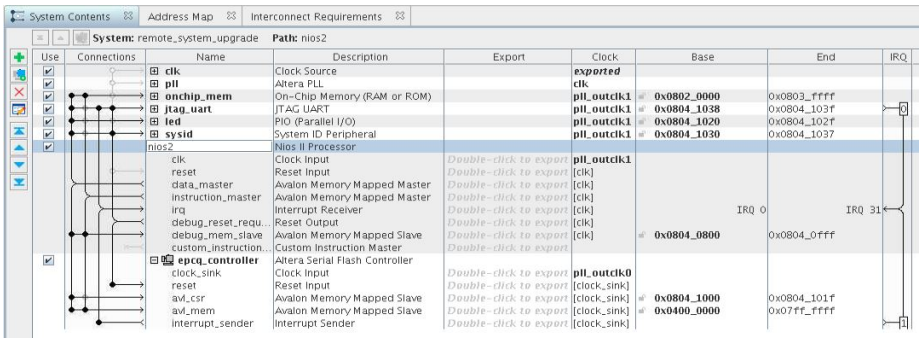
5.2.3.3.1. Hardware Design

The following sections describe a step-by-step method for building a bootable system for a Nios II processor application executing in place from EPCQ flash.

IP Component Settings

1. Create your Intel Quartus Prime and Platform Designer project.
2. Add the Intel FPGA Serial Flash Controller IP to your Platform Designer system. Refer to the diagram below for the IP connection in Platform Designer.

Figure 88. Intel FPGA Serial Flash Controller IP Connections in Platform Designer



Note:

- The maximum input clock for Intel FPGA Serial Flash Controller IP is 25 MHz. The input clock must not exceed this maximum value.
- Both Intel FPGA Serial Flash Controller and Intel FPGA Serial Flash Controller II are designed to support Intel EPCQ devices. Intel FPGA Serial Flash Controller II is the newest version and supports higher performance. Intel recommends to use the Intel FPGA Serial Flash Controller II for your new designs.

Reset and Exception Vector Settings for Nios II Execute-In-Place

1. Under **Reset Vector** in the **Vectors** tab, select EPCQ (**epcq_controller_0.avl_mem**) in the **Reset vector memory** drop-down menu and type the reset vector offset in the **Reset vector offset** entry box. The reset vector must be the base address of your application. In this example, it is 0x01E00000.

Note: Your ***.sof** image size influences your reset vector offset configuration. The reset vector offset is the start address of the **.hex** file in EPCQ flash and it must point to a location after the ***.sof** image. You can determine the minimum reset vector offset by using the following equation:

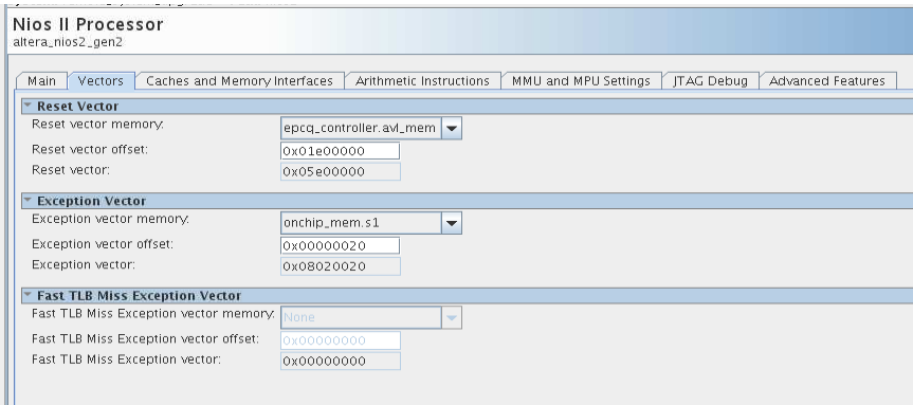
minimum reset vector offset= (**.sof** image start address + **.sof** image size) in HEX

For example, if your ***.sof** image starts at address 0x0 and is 512 KB in size, then the minimum reset vector offset location you can select is 0x0080000. If the ***.sof** image space and the reset vector offset location overlap, Intel Quartus Prime software displays an overlap error.

2. Under **Exception Vector**, you may select EPCQ (**epcq_controller_0.avl_mem**) or OCRAM in the **Exception vector memory** drop-down menu. In this example, 0x20 is listed for the **Exception vector offset** entry.

   Note: When executing-in-place, the Nios II processor boots and runs directly from EPCQ flash, without copying any code at boot time. Because the Nios II begins executing at the reset address in this case, exception vectors must be located at a nonzero exception vector offset to allow for instructions between the reset vector and the base of the exception vectors. Thus, you must include an exception vector offset when the reset vector and the exception vector point to the same memory and no boot copier is present.

Figure 89. Exception Vector Settings in Platform Designer



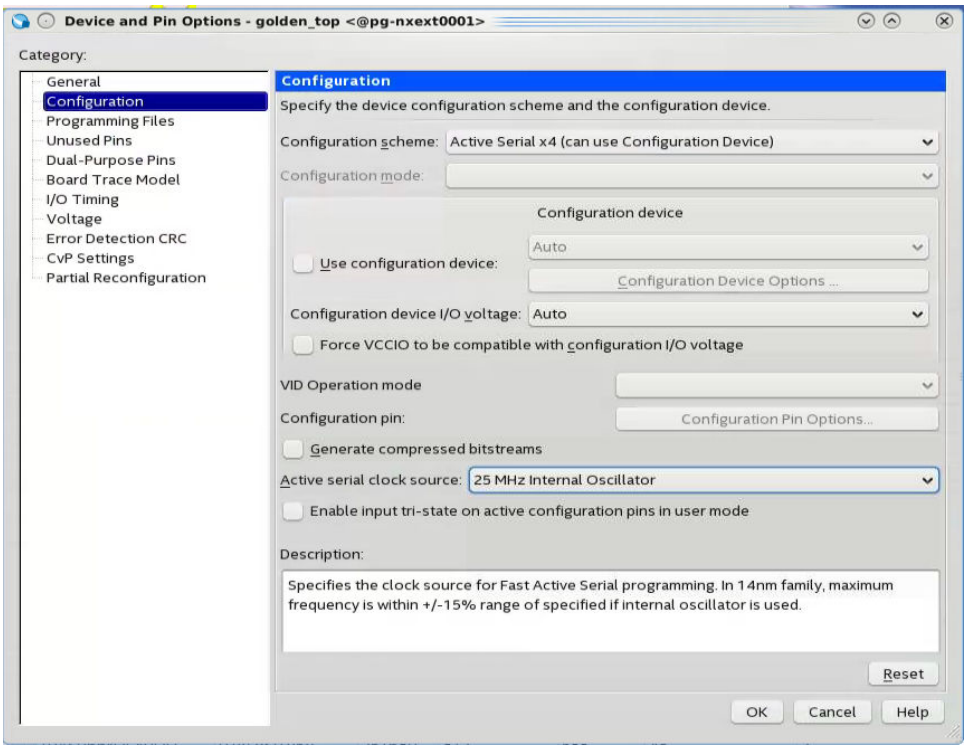
Note: Intel FPGA recommends setting the exception vector to OCRAM to make the interrupt processing faster.

3. Click **Finish**. You will return to the Platform Designer **System Contents** tab.
4. Double-click on the **Intel FPGA Serial Flash Controller IP** to open the **Intel FPGA Serial Flash Controller Parameter** editor.
5. Select the **Configuration device** type based on your hardware design and choose the desired **I/O mode**.
6. Close the **Parameter Editor** and return to the **Platform Designer System Contents** tab.
7. Click **Generate HDL** to generate your Platform Designer design.
8. Compile your design in Intel Quartus Prime software.

Intel Quartus Prime Software Settings

1. In the Intel Quartus Prime software, click on **Assignment ➤ Device ➤ Device and Pin Options ➤ Configuration**.
2. Set **Configuration scheme** to **Active Serial x4 (can use Configuration Device)**.
3. Set the **Active serial clock source** to **25 MHz Internal Oscillator**.

Figure 90. Device and Pin Options



5.2.3.3.2. Application

In the Nios II SBT window, select either:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.

  or

- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

BSP Editor Settings

1. Open the Nios II SBT tool and create the Nios II processor HAL BSP based on the `.sopinfo` created from Platform Designer generation.
2. Right-click on the BSP that you have created and select **Nios II ➤ BSP Editor**.
3. Configure the **Settings.Advanced.hal.linker** and the **Linker Section** mappings in the **BSP Editor** based on your boot options. Refer to the table below.

Table 34. BSP Editor Mappings

Boot Option	BSP Editor: Linker Script	Settings.Advanced.hal.linker
Nios II processor application executes in place from EPCQ flash	• Set `.text` Linker Section to EPCQ (`epcq_controller_0_avl_mem`) • Set other Linker Sections (`.heap`, `.rwdata`, `.rodata`, `.bss`, `.stack`) to OCRAM *Note:* If there is no OCRAM, then external RAM is used.	If the exception vector is set to OCRAM, enable the following settings in **Settings.Advanced.hal.linker**: • **allow_code_at_reset** • **enable_alt_load** • **enable_alt_load_copy_rodata** • **enable_alt_load_copy_rwdata** • **enable_alt_load_copy_exceptions** If the exception vector is set to EPCQ, enable the following settings in **Settings.Advanced.hal.linker**: • **allow_code_at_reset** • **enable_alt_load** • **enable_alt_load_copy_rodata** • **enable_alt_load_copy_rwdata**

Figure 91. Settings.Advanced.hal.linker for Execute-In-Place Method if Exception Vector is Set to OCRAM

This figure displays the settings if the Exception Vector is set to OCRAM.

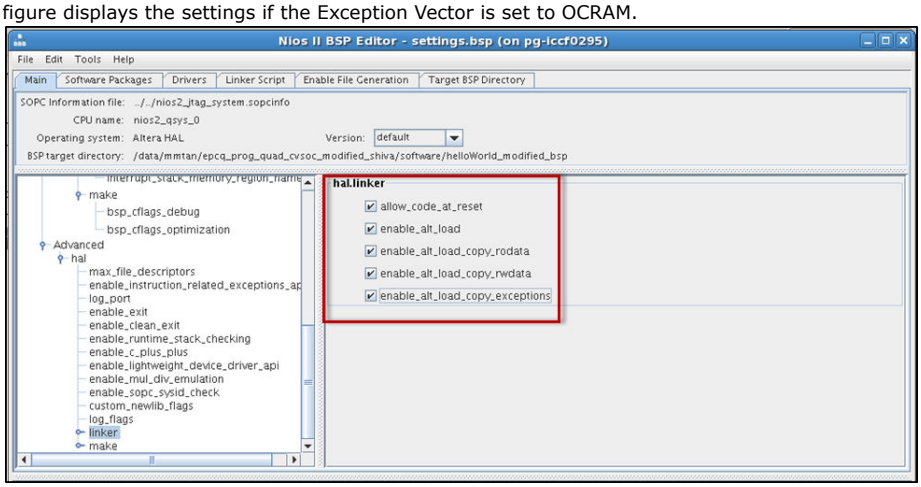


Figure 92. Settings.Advanced.hal Linker for Execute-in-Place Method if Exception Vector is Set to EPCQ

This figure displays the settings if the Exception Vector is set to EPCQ.

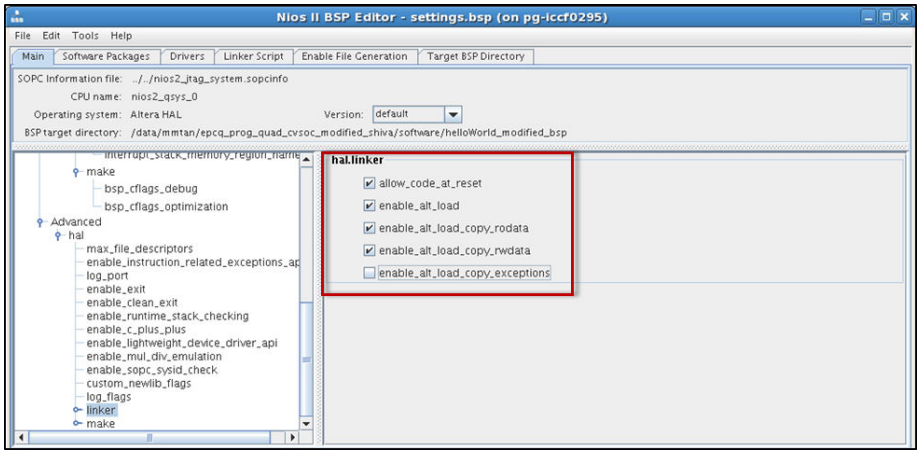
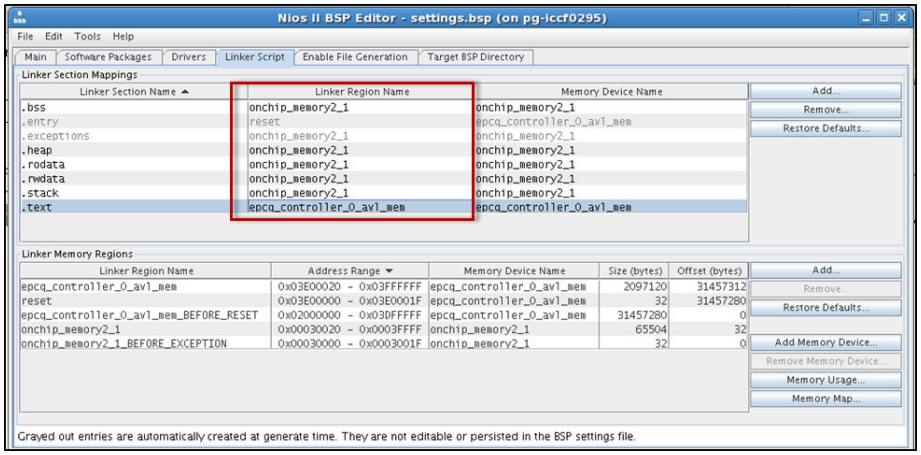


Figure 93. Linker Section Mappings for Execute-In-Place Method

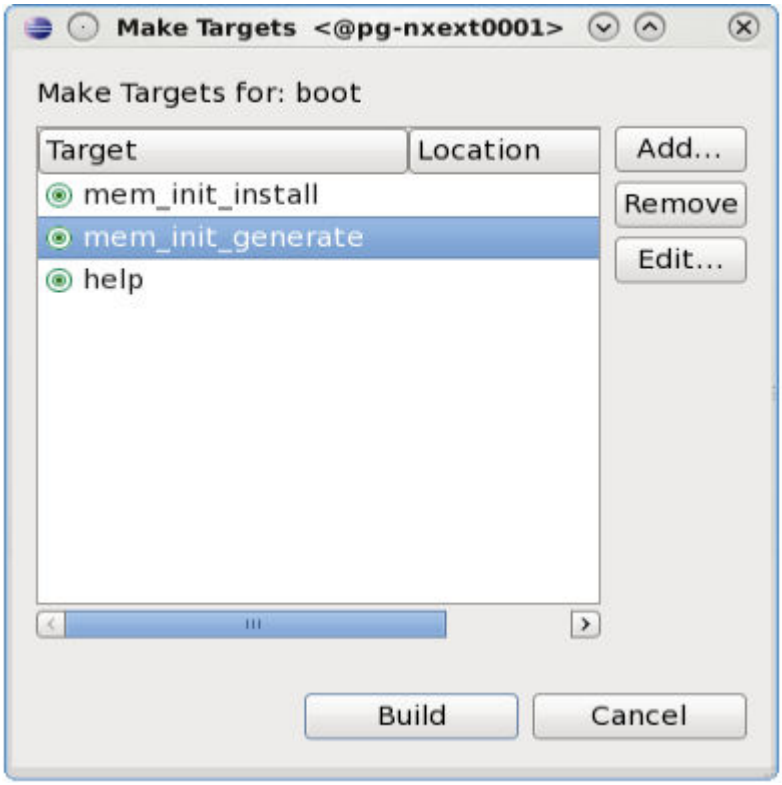


4. Click **Generate**. Make sure the BSP generation is successful.
5. Click **Exit** to close the BSP editor.

HEX File Generation

1. Point your application to the BSP location that you have created in the previous section.
2. Once the application development finishes, right-click on the project in **Project Explorer** and select **Build Project**. A ***.elf** file is created under the project folder.
3. Next, generate the HEX file using the mem_init_generate utility. Right-click on the project in Project Explorer and select **Make Targets > Build**. The Make Targets window opens as shown below.

Figure 94. Make Targets Window



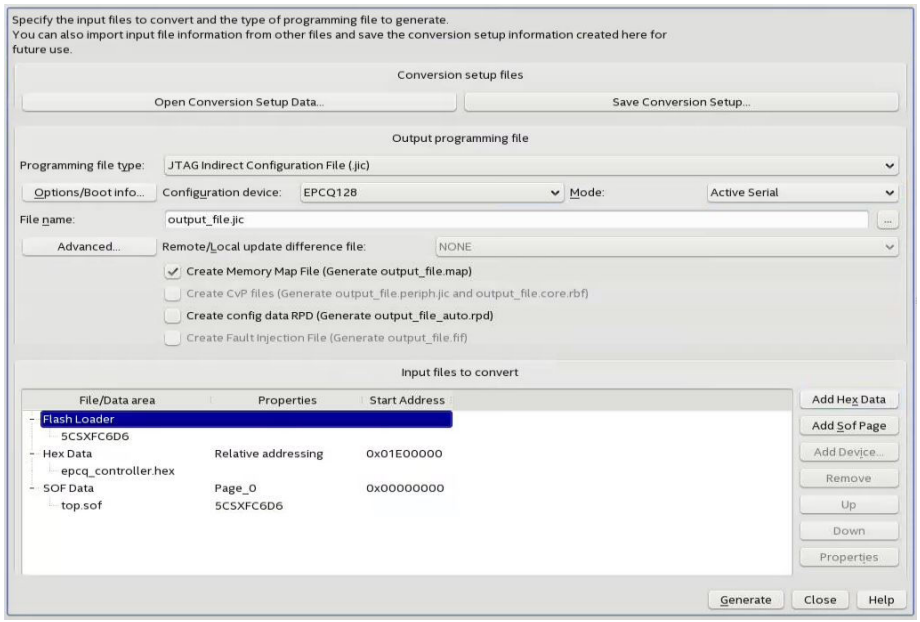
4. Select **mem_init_generate** and click **Build**. Make sure epcq_controller_0.hex is created under the mem_init folder.

5.2.3.3.3. Programming

Programming Files Generation

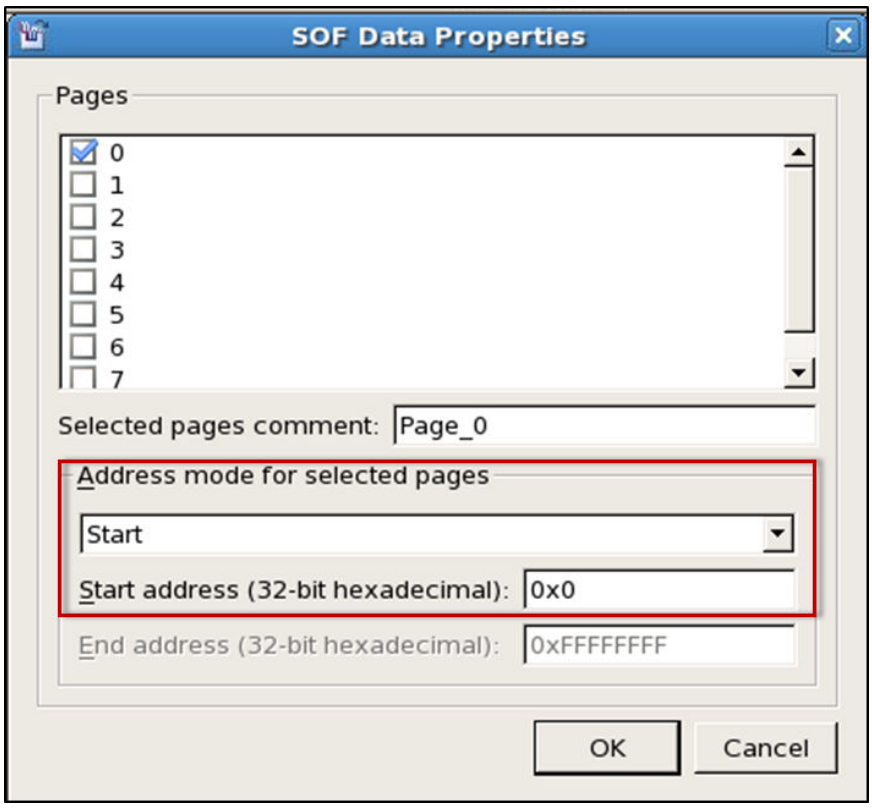
1. In the Intel Quartus Prime software, go to **File ➤ Convert Programming Files** to open the **Convert Programming File** tool.
2. Under **Output programming file** section, set the following items:
   a. **Programming file type:** JTAG Indirect Configuration File (`.jic`)
   b. **Configuration device:** Select according to your EPCQ type
   c. **Mode:** Active Serial
   d. **File name:** You may select your preferred path for the output file (`.jic`). By default this is generated under your project root directory.
   e. Keep the default settings for **Create Memory Map File** and **Create config data RPD**.

Figure 95. Convert Programming File Settings



3. Under **Input files to convert** section:
   a. Select **Flash Loader**, then **Add Device** to select the FPGA device that you are using. Click **OK** when you are done.
   b. Select **SOF Data**, then **Add File** to choose the `*.sof` file generated by Intel Quartus Prime compilation.
   c. Click on the `*.sof` file that you have just added. Next, select **Properties** and enable **Compression**.
   d. Click on **SOF Data** and select **Properties**. The **SOF Data Properties** window opens. Under **Address mode for selected pages**, select **Start** and set the start address (32-bit hexadecimal).

Figure 96. SOF Data Properties

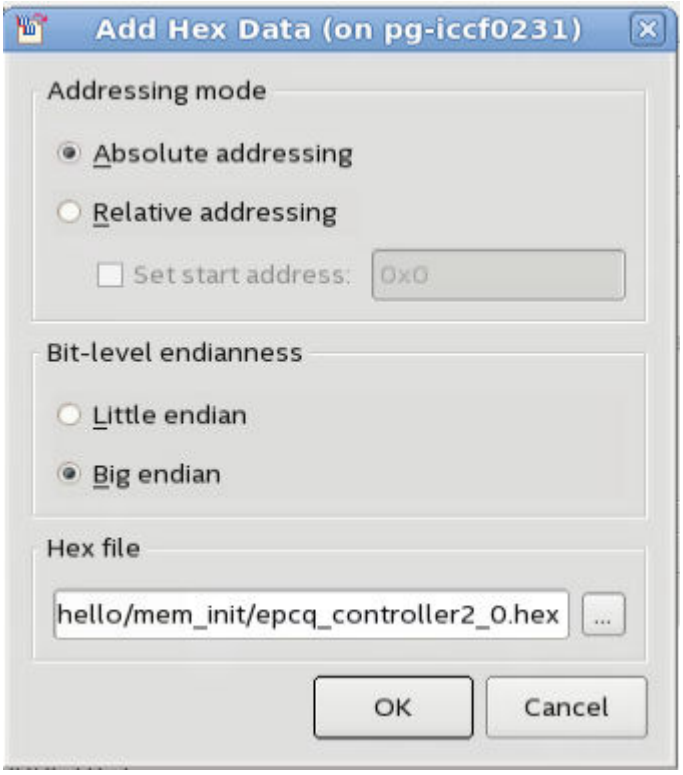


e. Click **OK** to close the window.

f. Select **Add Hex Data** to open the **Hex Data Properties** window. Select the **Absolute addressing** button and browse to the HEX file location. Click **OK** when you are done.

Note:
- You may select **Relative addressing** if you would like to set a relative address to the reset vector offset (0x01E00000) you configured earlier. For example, setting a start address of 0x01F00000 for the relative addressing mode changes the start address to 0x3D00000.
- There is an issue for Absolute Addressing for EPCQ. Refer to this knowledge database for more details.

Figure 97. HEX Data Properties



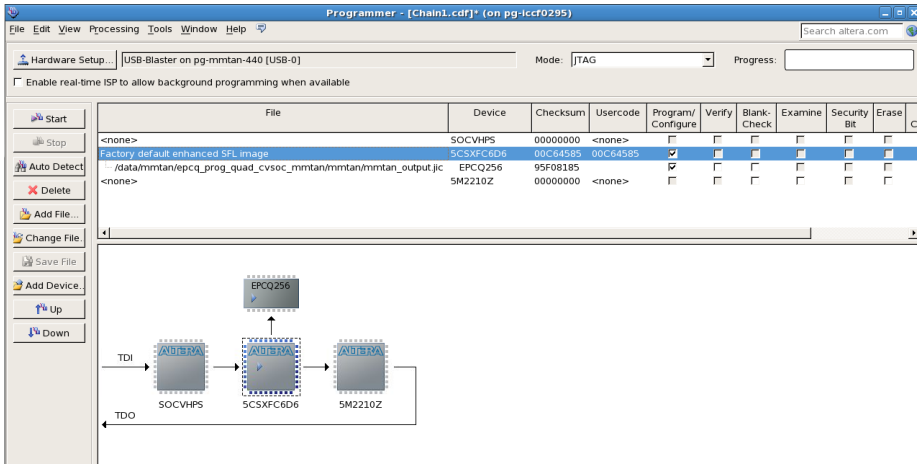
g. Select **Generate** to generate the `*.jic` programming file.

EPCQ Flash Programming

1. Ensure that the FPGA device's Active Serial (AS)[15] pin is routed to the EPCQ flash. This routing allows the flash loader to load into the EPCQ flash and configure the board correctly.
2. Ensure the MSEL pin setting on the board is configured for AS programming.
3. Open the Intel Quartus Prime Programmer and make sure JTAG was detected under the **Hardware Setup**.
4. Select **Auto Detect** and choose the FPGA device according to your board.
5. Right-click on the selected FPGA device and select **Edit ➤ Change File**. Next, select the generated `*.jic` file.
6. Select the **Program/ Configure** check boxes for FPGA and EPCQ devices. Click **Start** to start programming.

[15] MSEL setting is required for AS configuration

Figure 98. Intel Quartus Prime Programmer Window When Programming EPCQ Flash on the Development Board



7. Reset your FPGA device using the reset button or power cycle your hardware.

5.2.3.4. Nios II Processor Application Copied from EPCQ Flash to RAM Using Boot Copier

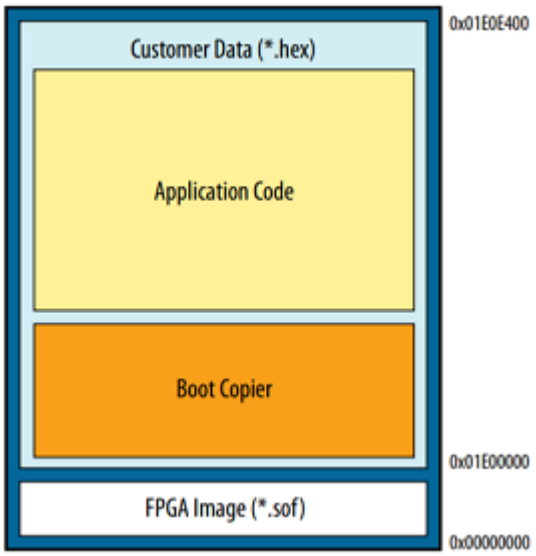
You can use a boot copier to copy the Nios II application from EPCQ flash to RAM when multiple iterations of application software development and high system performance are required. Intel FPGA recommends this solution for Nios II processor booting from Serial flash (EPCQ).

The Nios II SBT tool automatically adds the Nios II processor memcpy-based boot copier to the system when the executable file (**.elf**) is converted to the memory initialization file (**.hex**). The boot copier is located at the base address of the HEX data, followed by the application.

For this boot option, the Nios II processor starts executing the boot copier software upon system reset which copies the application from the EPCQ to the internal or external RAM. Once this completes, the Nios II processor transfers the program control over to the application.

Figure 99. EPCQ Flash Layout When Using Boot Copier

In this figure, the size of the FPGA image (*.**sof**) is estimated to be less than `0x01E00000`. Customer data (*.**hex**) is set to start at address `0x01E000000` and the size of the *.**hex** file is assumed to be `0x0000E400`.



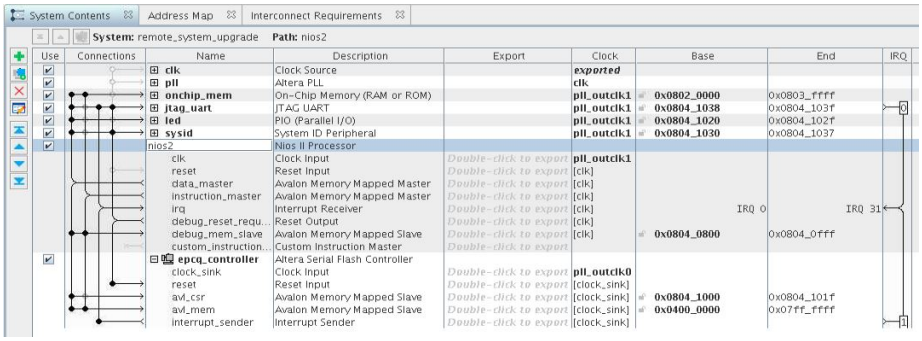
5.2.3.4.1. Hardware Design

The following sections describe a step-by-step method for building a bootable system for a Nios II processor application copied from QSPI flash to RAM using boot copier.

IP Component Settings

1. Create your Intel Quartus Prime and Platform Designer project.
2. Add the Intel FPGA Serial Flash Controller IP to your Platform Designer system. Refer to the diagram below for the IP connection in Platform Designer.

Figure 100. Intel FPGA Serial Flash Controller IP Connections in Platform Designer



Note: The maximum input clock for Intel FPGA Serial Flash Controller IP is 25 MHz. The input clock must not exceed this maximum value.

Both Intel FPGA Serial Flash Controller and Intel FPGA Serial Flash Controller II are designed to support Intel EPCQ devices. Intel FPGA Serial Flash Controller II is the newest version and supports higher performance. Intel recommends to use the Serial Flash Controller II for your new designs.

Reset and Exception Vector Settings for Nios II Boot Copier Method

1. Under **Reset Vector** in the **Vectors** tab, select EPCQ (**epcq_controller_0.avl_mem**) in the **Reset vector memory** drop-down menu and type the reset vector offset in the **Reset vector offset** entry box. The reset vector must be the starting address of your application. In this example, it is 0x01E00000.

   Note: Your `*.sof` image size influences your reset vector offset configuration. The reset vector offset is the start address of the `.hex` file in EPCQ flash and it must point to a location after the `*.sof` image. You can determine the minimum reset vector offset by using the following equation:

   minimum reset vector offset= (`.sof` image start address + `.sof` image size) in HEX

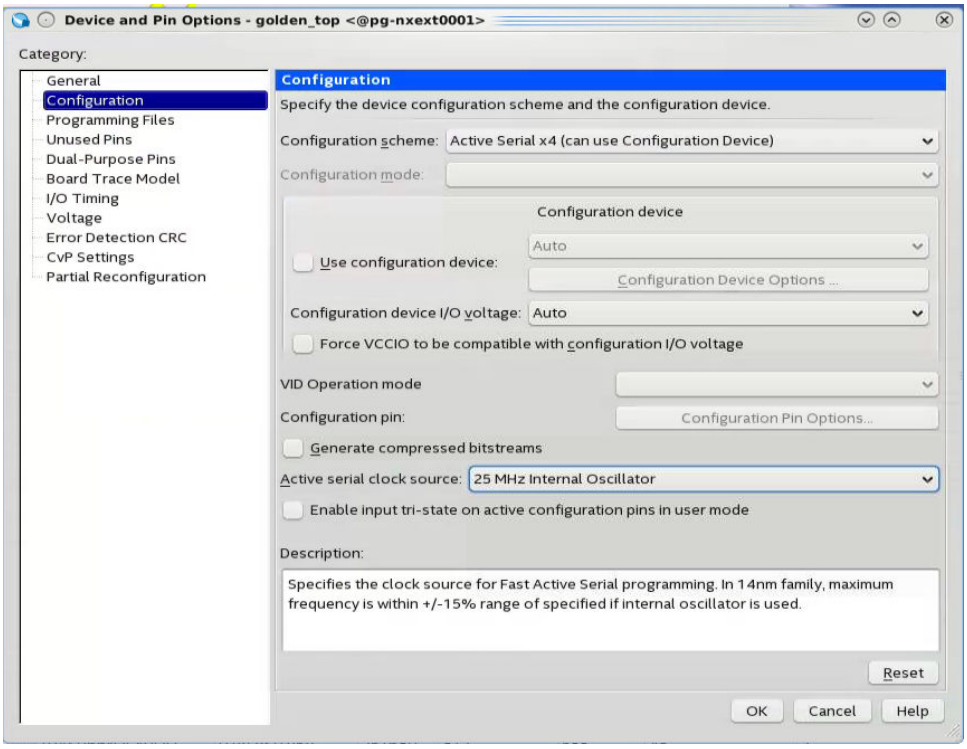
   For example, if your `*.sof` image starts at address 0x0 and is 512 KB in size, then the minimum reset vector offset location you can select is 0x0080000. If the `*.sof` image space and the reset vector offset location overlap, Intel Quartus Prime software displays an overlap error.
2. Under **Exception Vector**, select **OCRAM/External RAM** in the **Exception vector memory** drop-down menu. In this example, 0x20(16) is listed for the **Exception vector offset** entry.
3. Click **Finish**. You will return to the Platform Designer **System Contents** tab.
4. Double-click on the **Intel FPGA Serial Flash Controller IP** to open the **Intel FPGA Serial Flash Controller Parameter** editor.
5. Select the **Configuration device** type based on your hardware design and choose the desired **I/O mode**. Close the **Parameter Editor** and return to the **Platform Designer System Contents** tab.
6. Click **Generate HDL** to generate your Platform Designer design.
7. Compile your design in the Intel Quartus Prime software.

Intel Quartus Prime Software Settings

1. In the Intel Quartus Prime software, click on **Assignment ➤ Device ➤ Device and Pin Options ➤ Configuration**.
2. Set **Configuration scheme** to **Active Serial x4 (can use Configuration Device)**.
3. Set the **Active serial clock source** to **25 MHz Internal Oscillator**.

(16) If the reset vector and exception vector are pointing to the same memory, then 0x20 is the minimum offset for exception vector. This is to allocate 32 bytes for the `.entry` code section.

Figure 101. Device and Pin Options



5.2.3.4.2. Application

In the Nios II SBT window, select either:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.

  or

- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

BSP Editor Settings

1. Open the Nios II SBT tool and create the Nios II processor HAL BSP based on the `.sopinfo` created from Platform Designer generation.
2. Right-click on the BSP that you have created and select **Nios II ➤ BSP Editor**.
3. Configure the **Settings.Advanced.hal.linker** and the **Linker Section** mappings in the **BSP Editor** based on your boot options. Refer to the table below.

Table 35. BSP Editor Mappings

Boot Option	BSP Editor: Linker Script	Settings.Advanced.hal.linker
Nios II processor application is copied from EPCQ flash to RAM using boot copier	Make sure all Linker Sections are set to OCRAM/External RAM.	Make sure all settings are left unchecked.

Figure 102. Settings.Advanced.hal Linker for Boot Copier Method

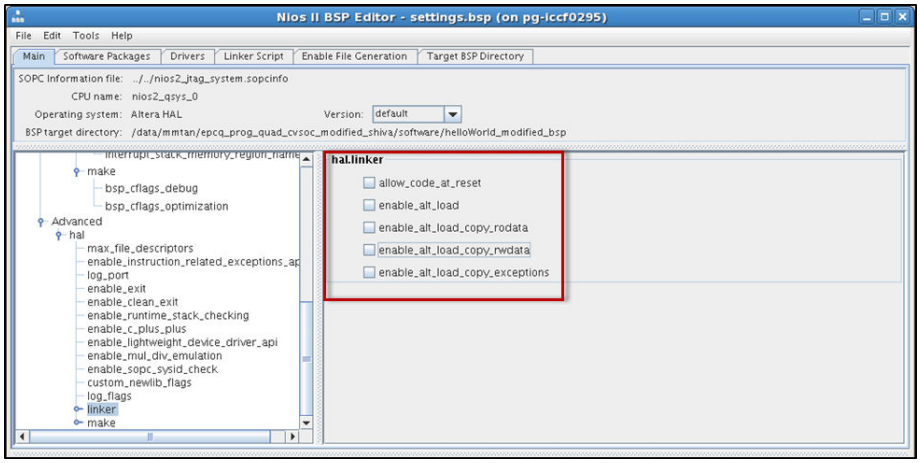
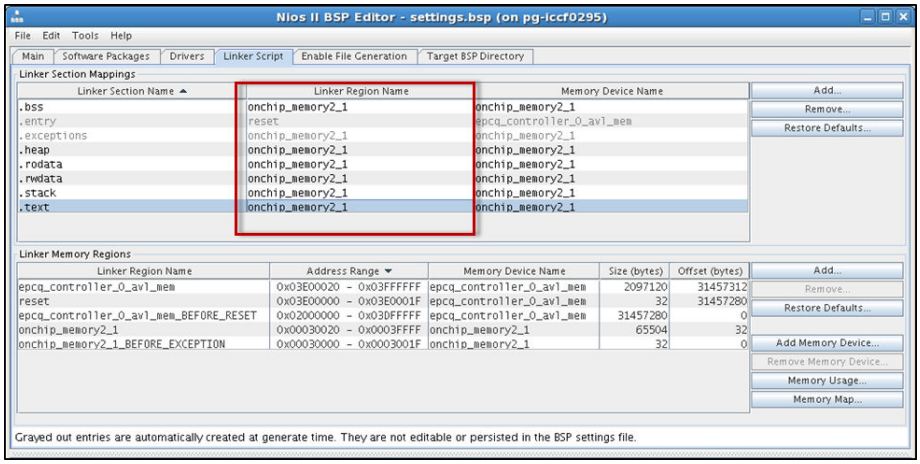


Figure 103. Linker Section Mappings for Boot Copier Method

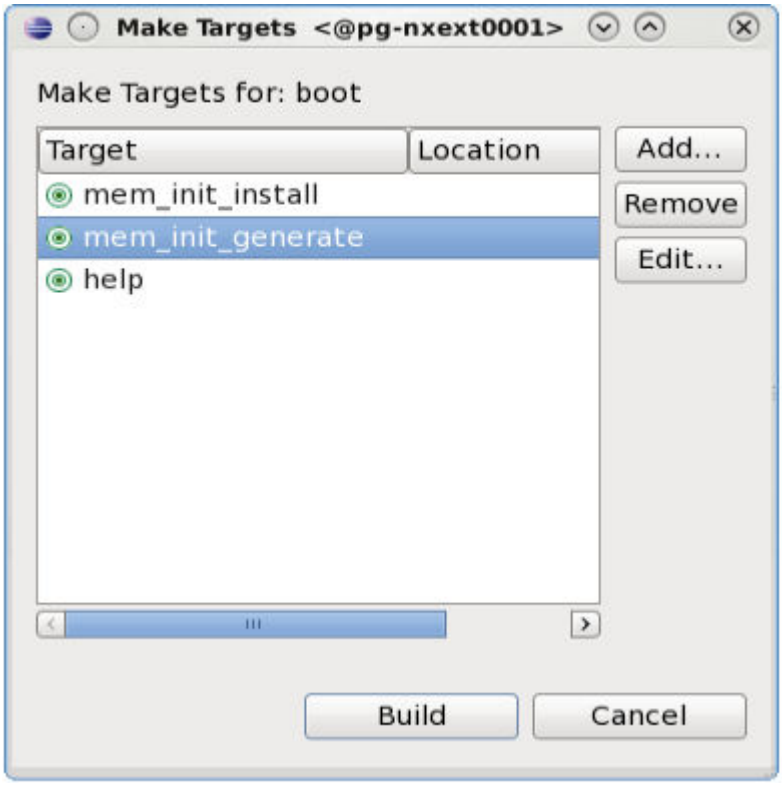


4. Click **Generate**. Make sure the BSP generation is successful.
5. Click **Exit** to close the BSP editor.

HEX File Generation

1. Point your application to the BSP location that you have created in the previous section.
2. Once the application development finishes, right-click on the project in **Project Explorer** and select **Build Project**. A ***.elf** file is created under the project folder.
3. Next, generate the HEX file using the mem_init_generate utility. Right-click on the project in Project Explorer and select **Make Targets > Build**. The Make Targets window opens as shown below.

Figure 104. Make Targets Window



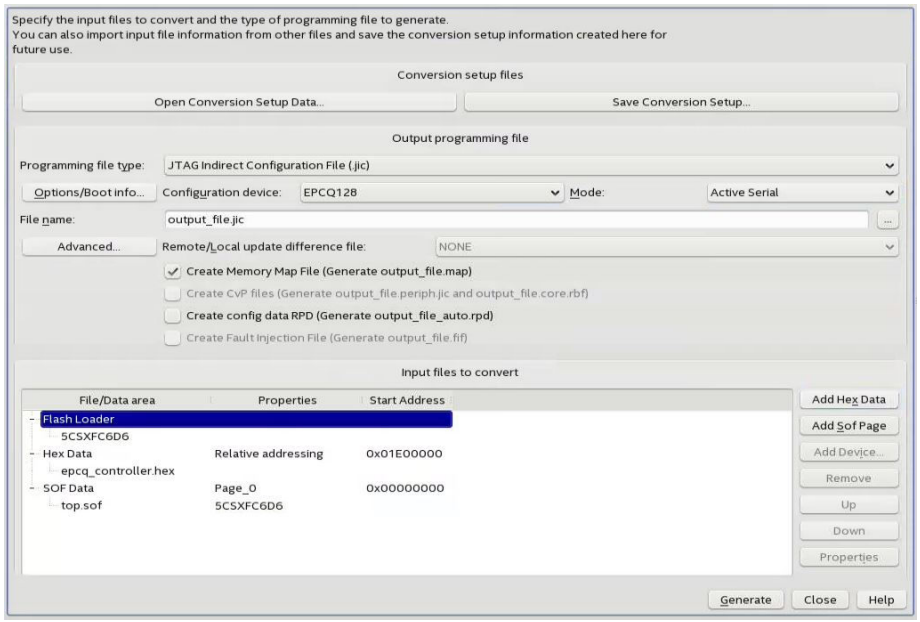
4. Select **mem_init_generate** and click **Build**. Make sure epcq_controller_0.hex is created under the mem_init folder.

5.2.3.4.3. Programming

Programming Files Generation

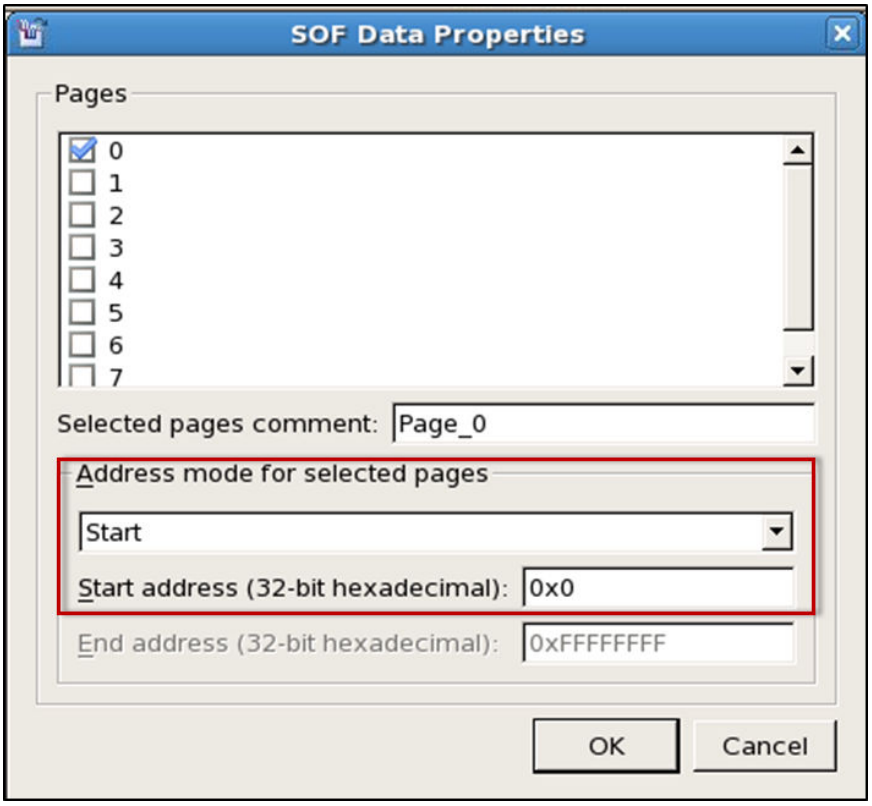
1. In the Intel Quartus Prime software, go to **File ➤ Convert Programming Files** to open the **Convert Programming File** tool.
2. Under **Output programming file** section, set the following items:
   a. **Programming file type:** JTAG Indirect Configuration File (`.jic`)
   b. **Configuration device:** Select according to your EPCQ type
   c. **Mode:** Active Serial
   d. **File name:** You may select your preferred path for the output file (`.jic`). By default this is generated under your project root directory.
   e. Keep the default settings for **Create Memory Map File** and **Create config data RPD**.

Figure 105. Convert Programming File Settings



3. Under **Input files to convert** section:
   a. Select **Flash Loader**, then **Add Device** to select the FPGA device that you are using. Click **OK** when you are done.
   b. Select **SOF Data**, then **Add File** to choose the `*.sof` file generated by Intel Quartus Prime compilation.
   c. Click on the `*.sof` file that you have just added. Next, select **Properties** and enable **Compression**.
   d. Click on **SOF Data** and select **Properties**. The **SOF Data Properties** window opens. Under **Address mode for selected pages**, select **Start** and set the start address (32-bit hexadecimal).

Figure 106. SOF Data Properties

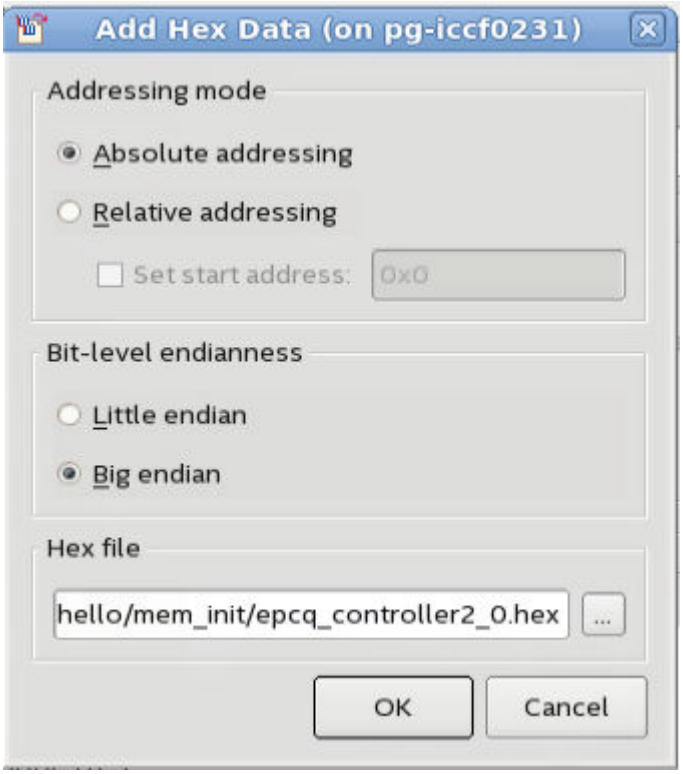


e. Click **OK** to close the window.

f. Select **Add Hex Data** to open the **Hex Data Properties** window. Select the **Absolute addressing** button and browse to the HEX file location. Click **OK** when you are done.

Note:
- You may select **Relative addressing** if you would like to set a relative address to the reset vector offset (0x01E00000) you configured earlier. For example, setting a start address of 0x01F00000 for the relative addressing mode changes the start address to 0x3D00000.

Figure 107. HEX Data Properties



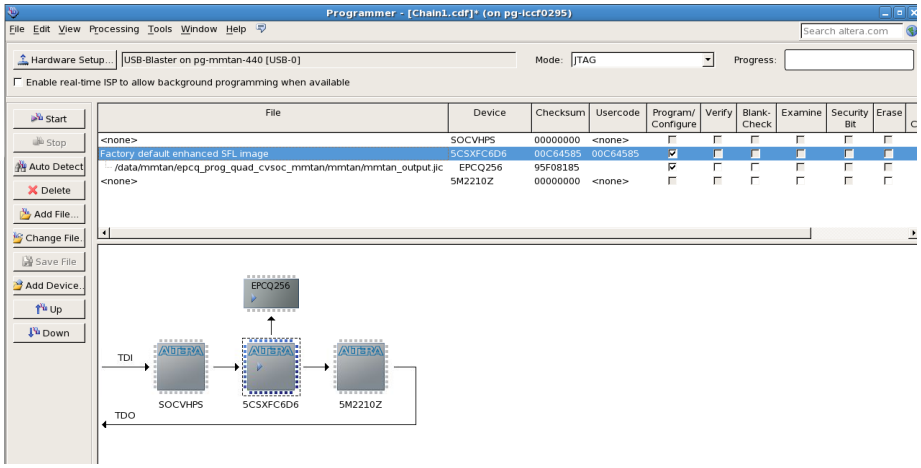
g. Select **Generate** to generate the `*.jic` programming file.

EPCQ Flash Programming

1. Ensure that the FPGA device's Active Serial (AS)[17] pin is routed to the EPCQ flash. This routing allows the flash loader to load into the EPCQ flash and configure the board correctly.
2. Ensure the MSEL pin setting on the board is configured for AS programming.
3. Open the Intel Quartus Prime Programmer and make sure JTAG was detected under the **Hardware Setup**.
4. Select **Auto Detect** and choose the FPGA device according to your board.
5. Right-click on the selected FPGA device and select **Edit ➤ Change File**. Next, select the generated `*.jic` file.
6. Select the **Program/ Configure** check boxes for FPGA and EPCQ devices. Click **Start** to start programming.

[17] MSEL setting is required for AS configuration

Figure 108. Intel Quartus Prime Programmer Window When Programming EPCQ Flash on the Development Board



7. Reset your FPGA device using the reset button or power cycle your hardware.

5.2.3.5. EPCQ HAL Driver

The Nios II hardware abstraction layer (HAL) driver has a generic application programming interface (API) for flash devices.

This API provides you with read and write access to the EPCQ flash through the Intel FPGA Serial Flash Controller for Intel Quartus Prime 15.0 or later. Use driver version 15.0 (or higher) for EPCQ software application development.

5.2.4. Nios II Processor Booting from QSPI Flash

The Nios II processor supports the following two boot options using QSPI flash:

- The Nios II processor application executes in place from QSPI flash.
- The Nios II processor application is copied from QSPI flash to RAM using a boot copier.

Note: For Intel MAX 10, the supported configuration modes are:

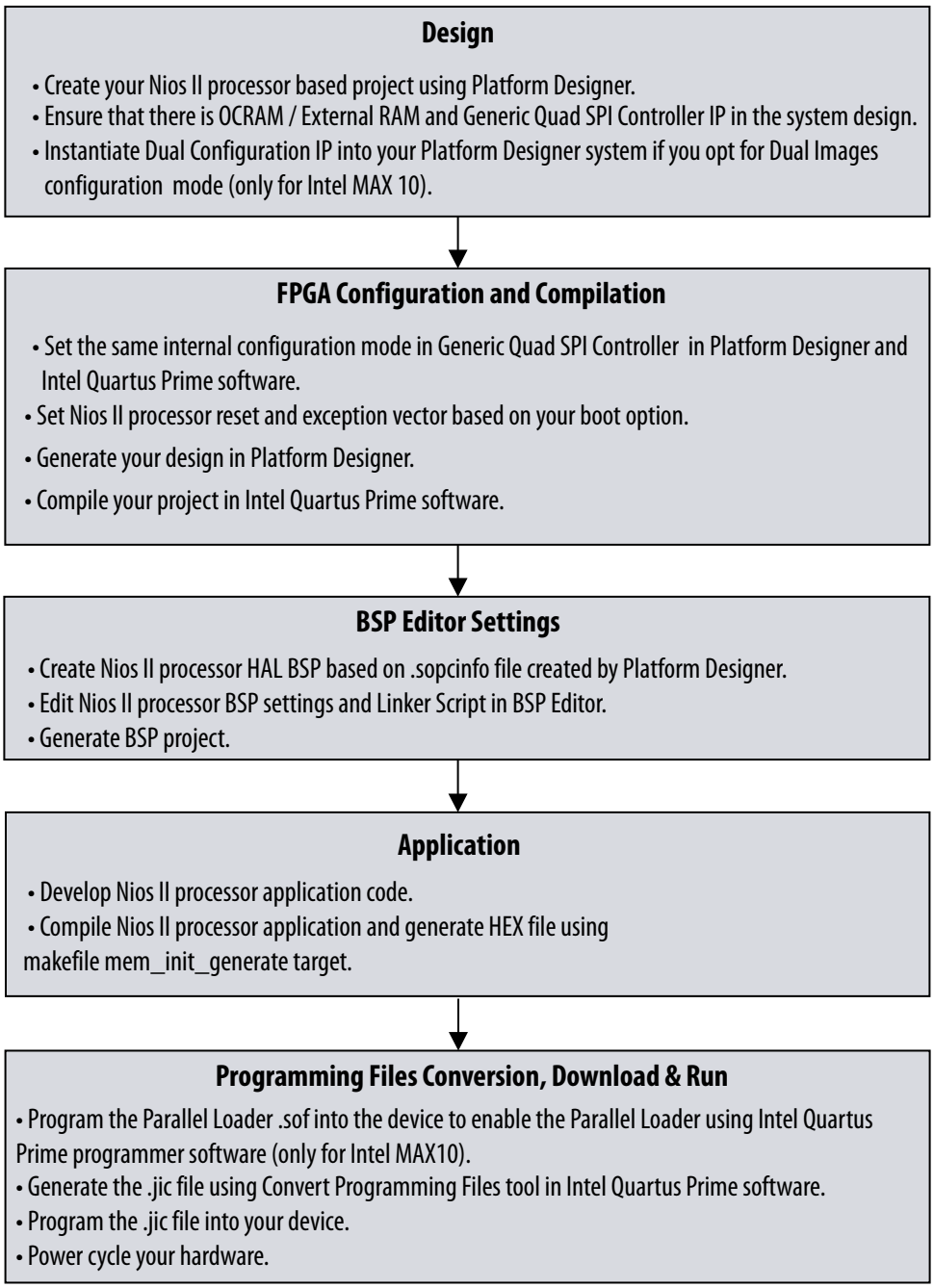
- Single Uncompressed Image
- Single Compressed Image
- Dual Compressed Images

Table 36. Summary of Nios II Processor Boot Options Using QSPI Flash

Boot Option	Application Code Stored Location	Application Run-time Location	Method
Nios II processor application execute-in-place from QSPI flash	QSPI flash	QSPI flash (XIP) + OCRAM/ External RAM (for writable data sections)	Enable `alt_load ()` function
Nios II processor application copied from QSPI flash to RAM using boot copier	QSPI flash	OCRAM/ External RAM	Memcpy-based boot copier

5.2.4.1. Nios II Processor Design, Configuration and Boot Flow

Figure 109. Design, Configuration and Booting Flow



5.2.4.2. Nios II Processor Application Executes In-Place from QSPI Flash

This is suitable for Nios II processor applications that requires code space larger than the UFM (Intel MAX 10 device only) and/or with a limited on-chip memory usage. This is an advantage for supporting larger or multiple software applications.

The `alt_load ()` function operates as a mini boot copier that initializes and copies the writable memory sections only to OCRAM or external RAM. The code section (`.text`), which is a read-only section, remains in the QSPI flash memory region. Retaining the read-only section in QSPI minimizes RAM usage but may limit the code execution performance.

The Nios II processor application is programmed into the QSPI flash. The Nios II processor reset vector points to the QSPI flash to allow code execution after the system resets. If you are debugging the application using the source-level debugger, you must use a hardware break-point because the QSPI cannot support random memory access.

Figure 110. Nios II Application XIP from QSPI flash (for Intel MAX 10)

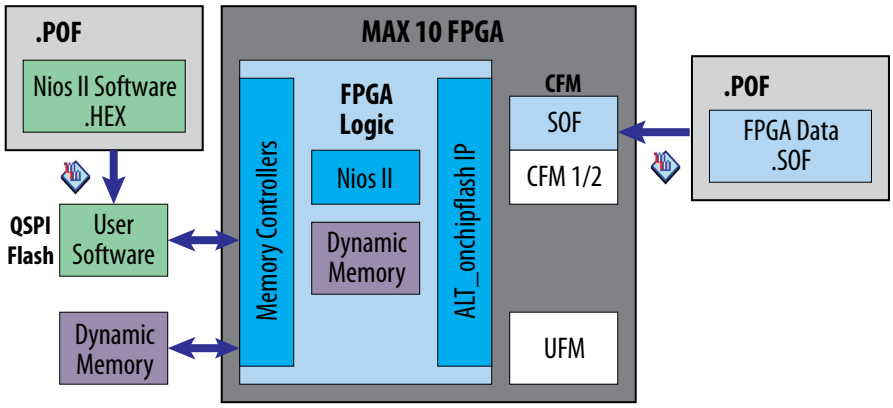


Table 37. RAM and ROM Size Requirement

RAM Size Requirement	ROM Size Requirement
Equivalent to the dynamic memory space usage during run time which is the sum of the maximum heap and stack size.	Executable code must not exceed the size of the QSPI flash.

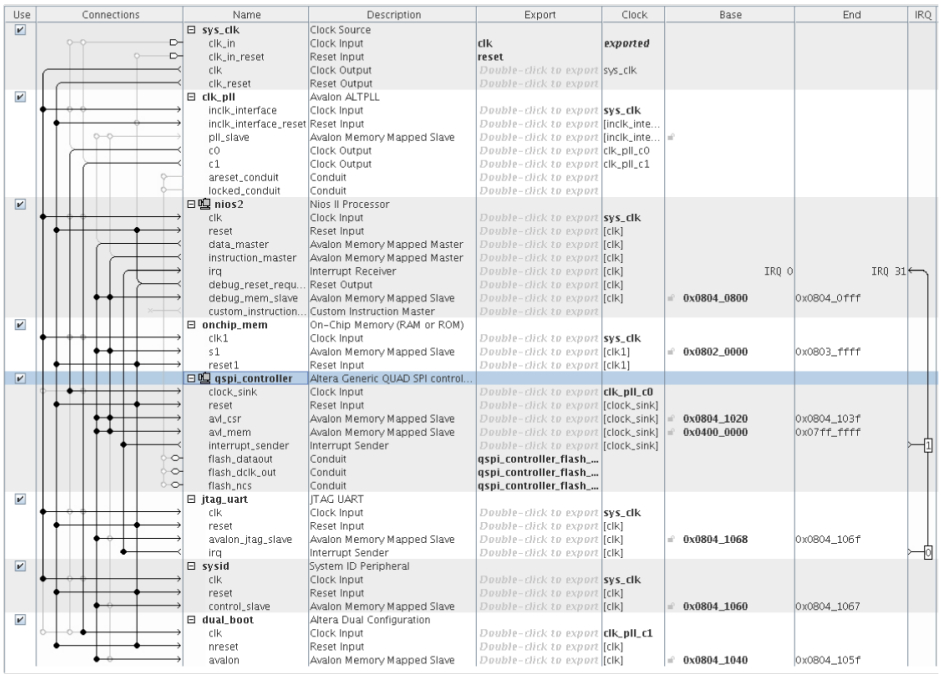
5.2.4.2.1. Hardware Design

The following sections describe a step-by-step method for building a bootable system for a Nios II processor application executing in place from QSPI flash.

IP Component Settings

1. Create your Nios II Processor Project using Intel Quartus Prime and Platform Designer.
2. Ensure the Generic Quad SPI Controller IP is added into your Platform Designer system.

Figure 111. Connection for Generic Quad SPI Controller IP



Note: Both Generic QUAD SPI Controller and Generic QUAD SPI Controller II are designed to support serial flash devices from third-party vendors. Generic QUAD SPI Controller II is the newest version and supports higher performance. Intel recommends to use the Generic QUAD SPI Controller II for your new designs.

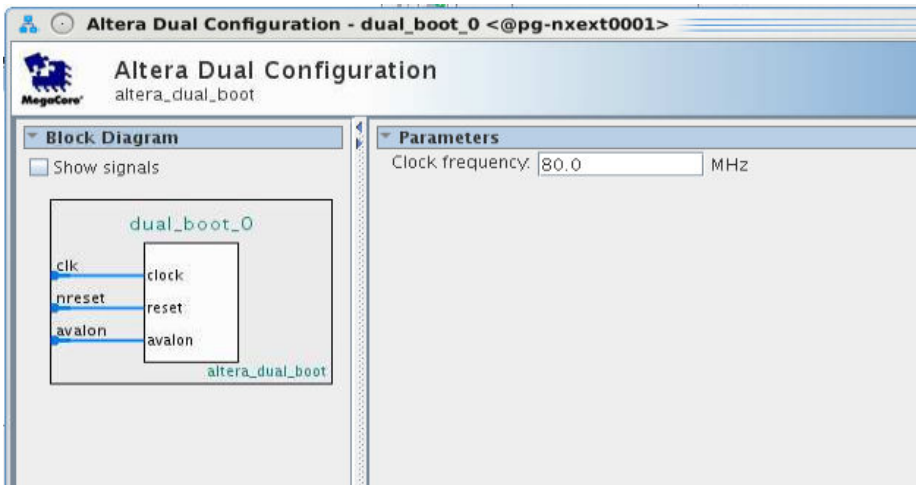
Important: If you use a serial flash device that is not supported by Generic Quad SPI Controller II, Intel recommends to use Generic Serial Flash Interface IP.

- Generic Serial Flash Interface Intel FPGA IP User Guide

3. If you want to enable dual images configuration, ensure that Dual Configuration IP is instantiated in your Platform Designer system.

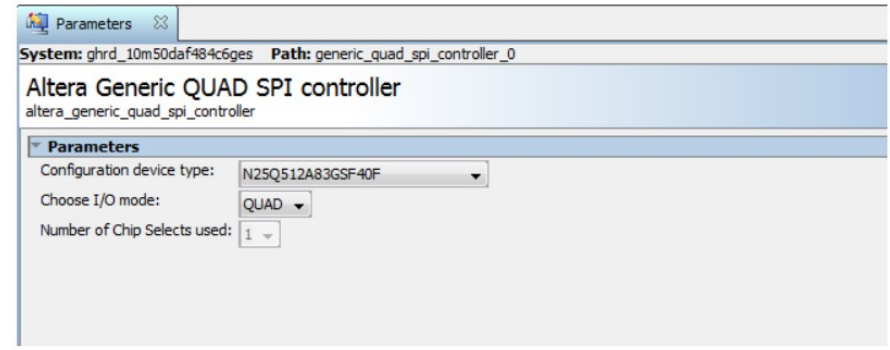
   Note: Dual Images Configuration is only available in Intel MAX 10 devices

Figure 112. Dual Configuration IP



4. Open Generic Quad SPI controller parameter editor. Change the Configuration device type to the QSPI flash selection and make sure the I/O mode is set to QUAD.

Figure 113. Generic Quad SPI Controller Parameter Settings

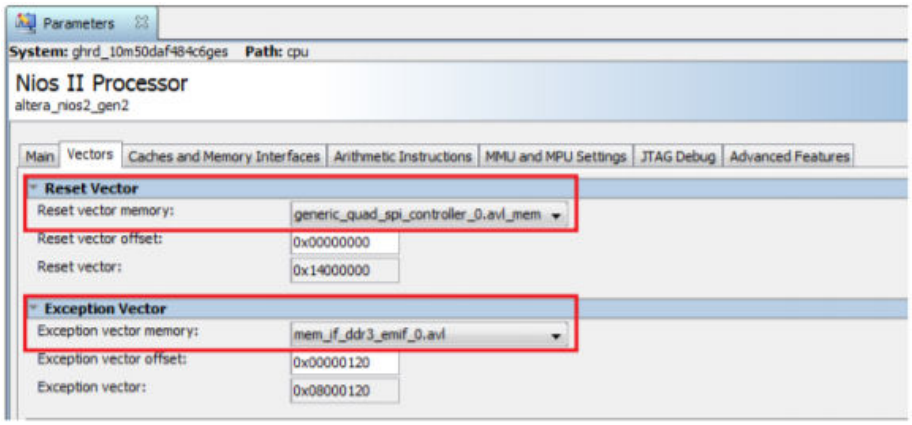


5. Click Generate HDL, the Generation dialog box appears.
6. Specify output file generation options and then click **Generate**.

Reset and Exception Vector Settings for Nios II Execute-In-Place Method

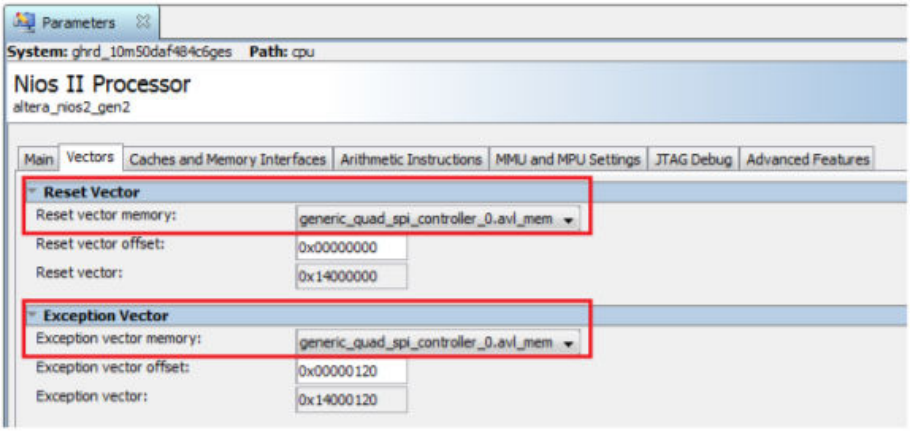
1. In the Nios II Processor parameter editor, set the reset vector memory to QSPI Flash and exception vector memory to OCRAM/ External RAM or QSPI Flash according to your design preference.

Figure 114. Nios II Parameter Editor Settings When Exception Vector is set to OCRAM/ External RAM



Note: `mem_if_ddr3_emif_0.avl` is the external memory (DDR3) used in this example.

Figure 115. Nios II Parameter Editor Settings When Exception Vector is set to QSPI Flash



2. For dual configuration images in Intel MAX 10 devices, there are two Nios II application data (HEX file) stored into the QSPI. Reset vector memory offset has to set correctly for the configuration images to call up the correct HEX data.
3. Set Reset vector memory offset of the Nios II Processor in first Platform Designer design to address `0x00000000`.
4. Set Reset vector memory offset of the Nios II Processor in second Platform Designer design to another address to avoid overlapping. For example: Address `0x02000000` which is half of the QSPI memory size (512 Mb).

Figure 116. Reset Vector Offset Setting for First Platform Designer Design

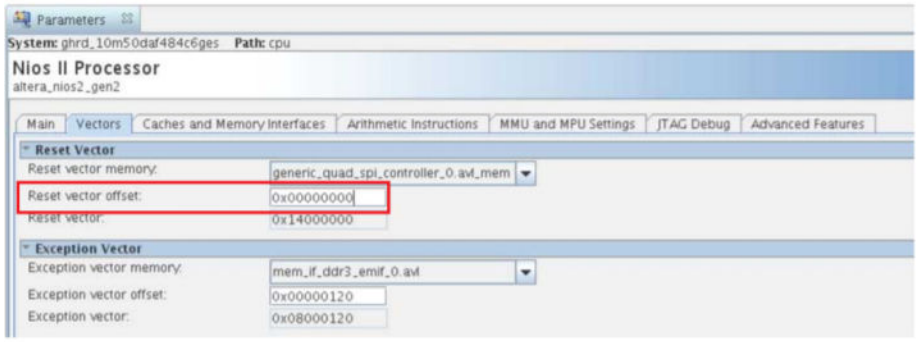
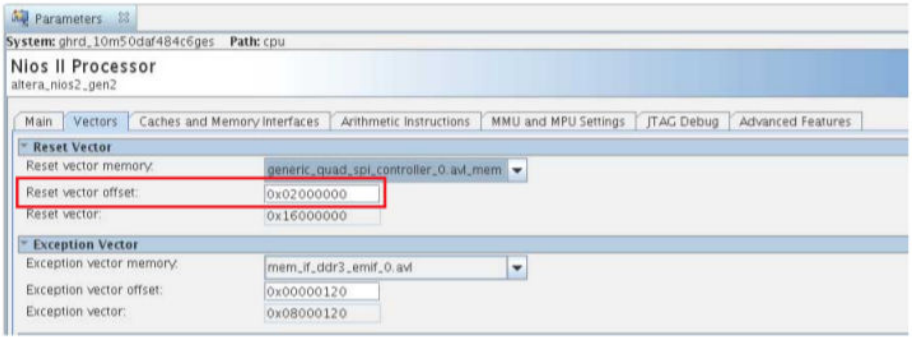


Figure 117. Reset Vector Offset Setting for Second Platform Designer Design

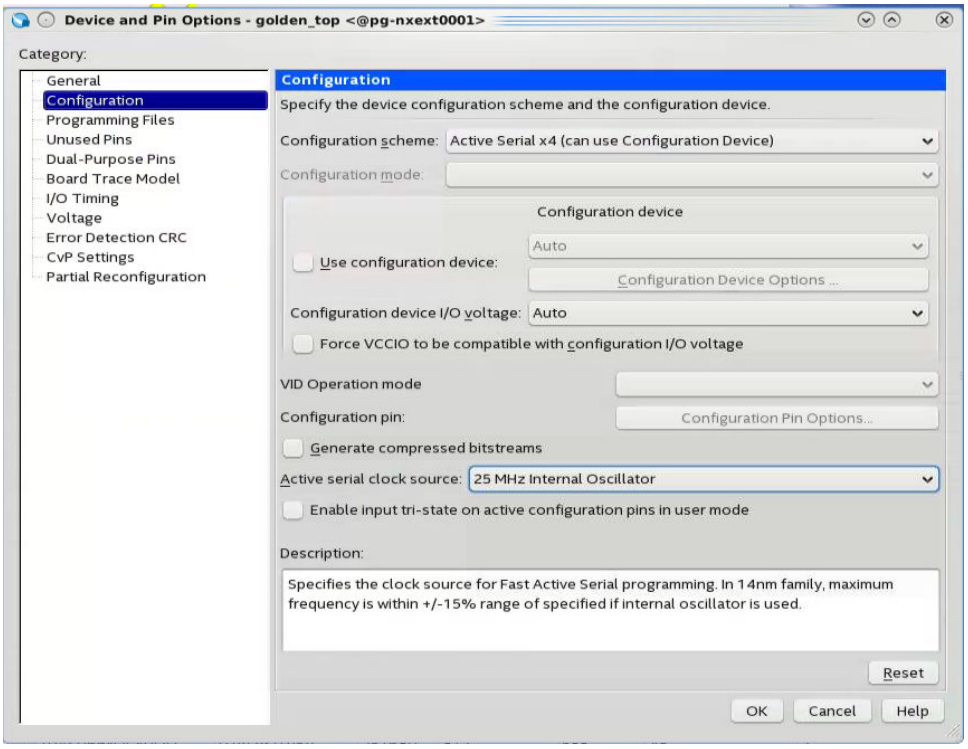


Intel Quartus Prime Software Settings

For non-Max10 devices:

1. In the Intel Quartus Prime software, click on **Assignment ➤ Device ➤ Device and Pin Options ➤ Configuration**.
2. Set **Configuration scheme** to **Active Serial x4 (can use Configuration Device)**.
3. Set the **Active serial clock source** to **25 MHz Internal Oscillator**.

Figure 118. Device and Pin Options

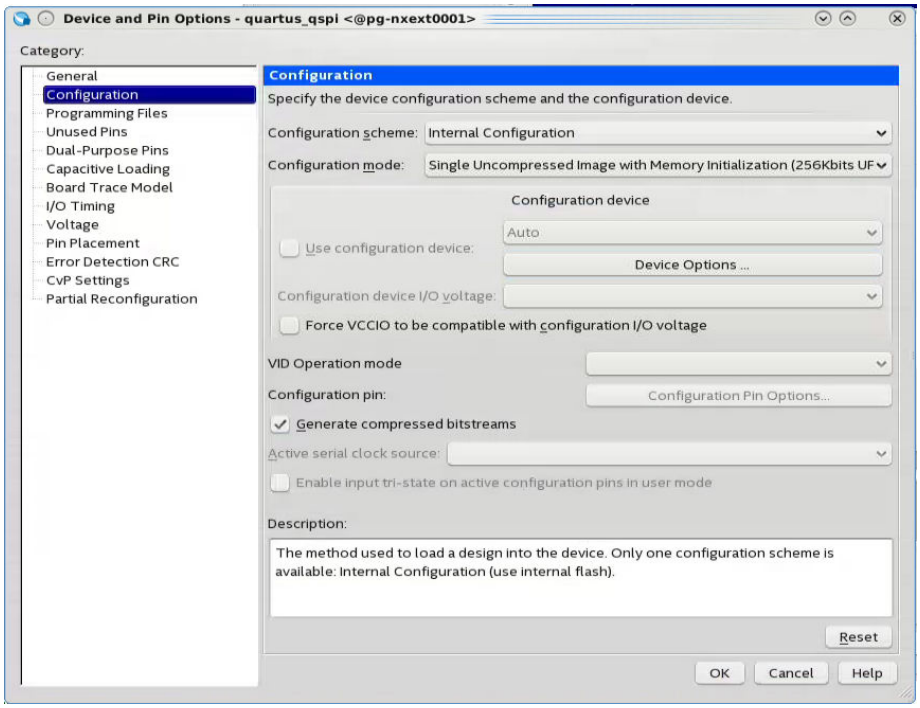


For Intel MAX 10 devices, there will be an option for Single Configuration Image and Dual Configuration Image:

1. In the Intel Quartus Prime software, click on **Assignment ➤ Device ➤ Device and Pin Options ➤ Configuration**. Set **Configuration mode** to **Single Uncompressed Image** or **Single Compressed Image**.

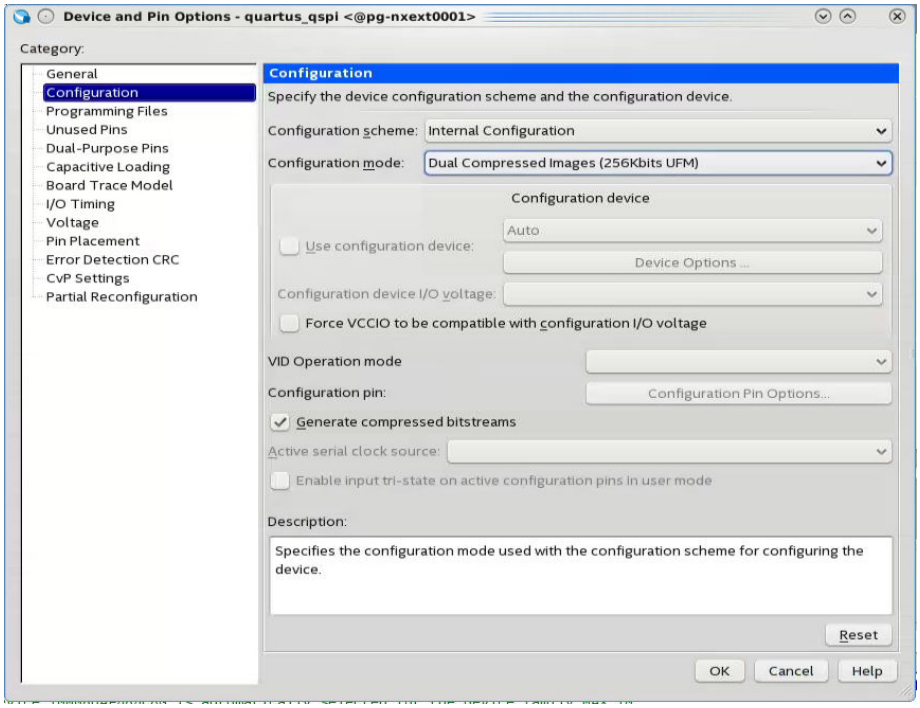
   Note: Intel MAX 10 only supports Single Uncompressed Image with Memory Initialization

Figure 119. Single Configuration Mode Selection in Intel Quartus Prime Software



2. If you want to set it to Dual Configuration Image, in the Intel Quartus Prime software, click on **Assignment -> Device -> Device and Pin Options -> Configuration**. Set **Configuration mode** to **Dual Compressed Images**.

Figure 120. Dual Configuration Mode Selection in Quartus Prime Software



3. Click **OK** to exit the **Device and Pin Options** window.
4. Click **OK** to exit the **Device** window.
5. Click **Start Compilation** to compile your project.

   Note: SOF to POF file conversion is not required because there is only single hardware image and Nios II application data will be loaded into the QSPI flash separately. You can use the POF generated duringIntel Quartus Prime project compilation to program into the device.

5.2.4.2.2. Application

In the Nios II SBT window, select either:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.

  or

- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

BSP Editor Settings

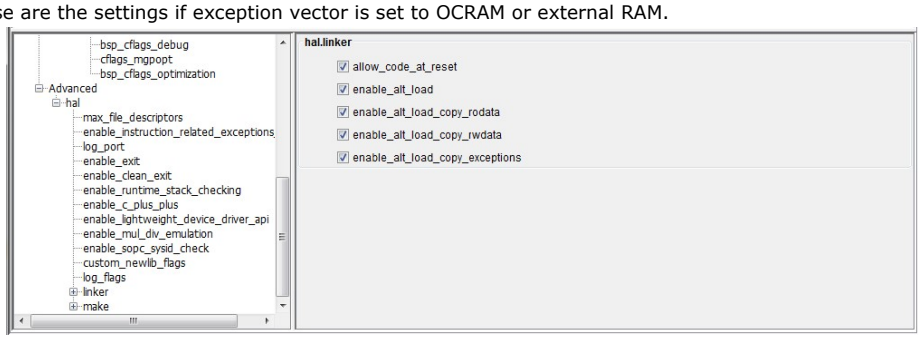
You must edit the BSP editor settings according to the selected Nios II processor boot options.

1. In the Nios II SBT tool, right click on your BSP project in the **Project Explorer** window. Select **Nios II** > **BSP Editor...** to open the **Nios II BSP Editor**.
2. In Nios II BSP Editor, expand into **Advanced.hal.linker** under **Settings**.
3. If exception vector memory is set to OCRAM or External RAM, enable the following:

- **allow_code_at_reset**
- **enable_alt_load**
- **enable_alt_load_copy_rodata**
- **enable_alt_load_copy_rwdata**
- **enable_alt_load_copy_exceptions**

Figure 121. Advanced.hal.linker Settings

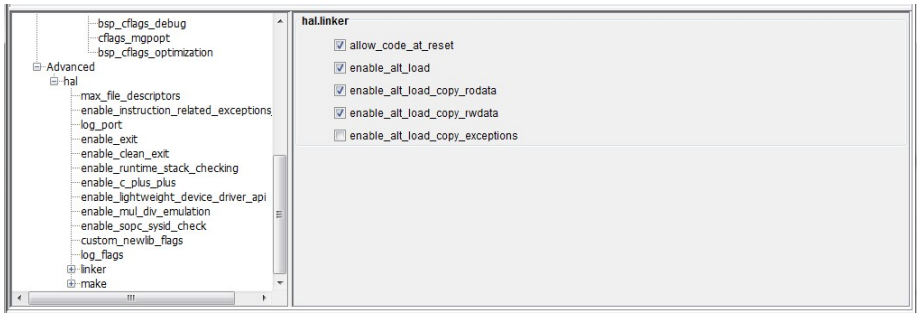
These are the settings if exception vector is set to OCRAM or external RAM.



4. If exception vector memory is set to QSPI flash, enable the following:
   - **allow_code_at_reset**
   - **enable_alt_load**
   - **enable_alt_load_copy_rodata**
   - **enable_alt_load_copy_rwdata**

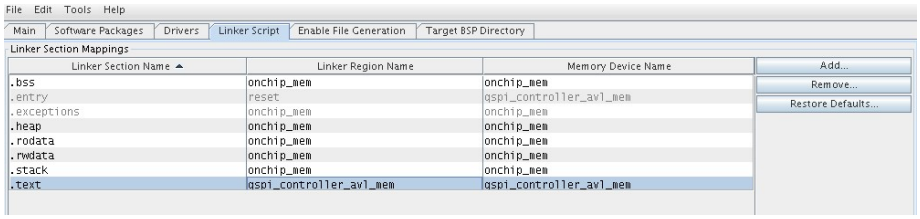
Figure 122. Advanced.hal.linker Settings

These are the settings if exception vector is set to QSPI flash.



5. Click on the **Linker Script** tab in the Nios II BSP Editor.
6. Set the .text item in the **Linker Section Name** to the QSPI flash in the **Linker Region Name**. Set the rest of the items in the **Linker Section Name** list to the Onchip Memory (OCRAM) or external RAM.

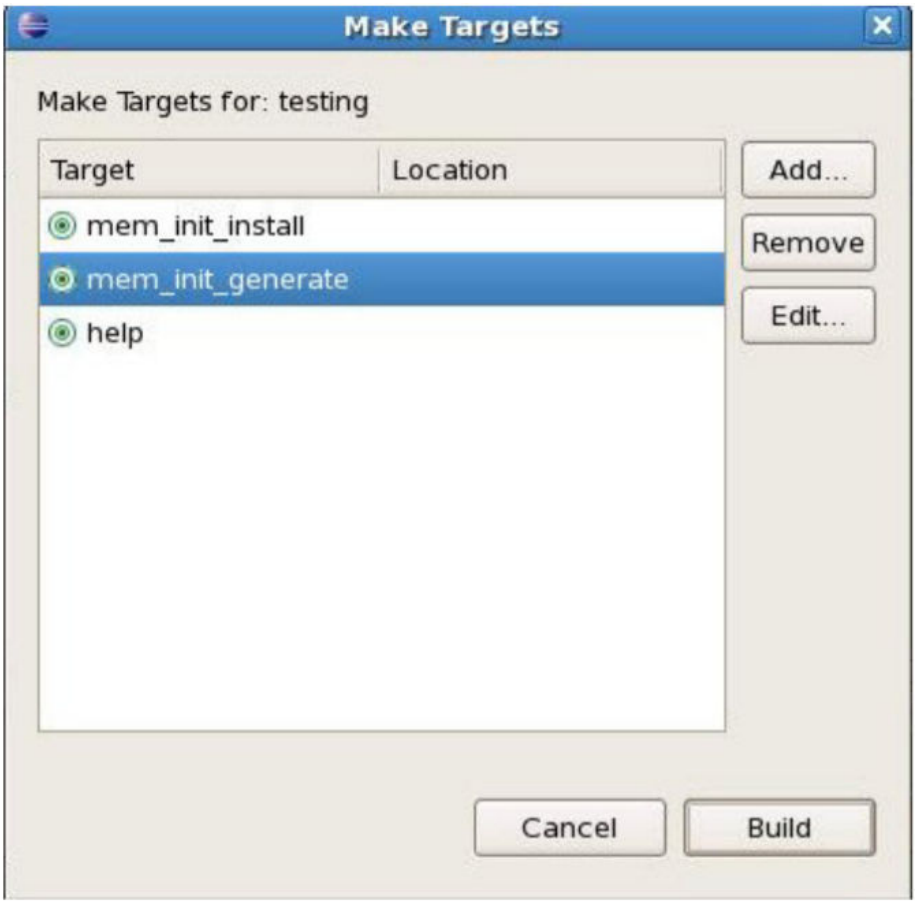
Figure 123. Linker Region Settings



HEX File Generation

1. In the **Nios II SBT tool**, right click on your project in the **Project Explorer** window.
2. Click **Make Targets** -> **Build...**, the **Make Targets** dialog box appears.
3. Select **mem_init_generate**.
4. Click **Build** to generate the HEX file.

Figure 124. Selecting `mem_init_generate` in Make Targets



5. The "mem_init_generate" macro will create two HEX files; <OCRAM_name>.hex and <QSPIFlash_name>.hex. Select <QSPIFlash_name>.hex.

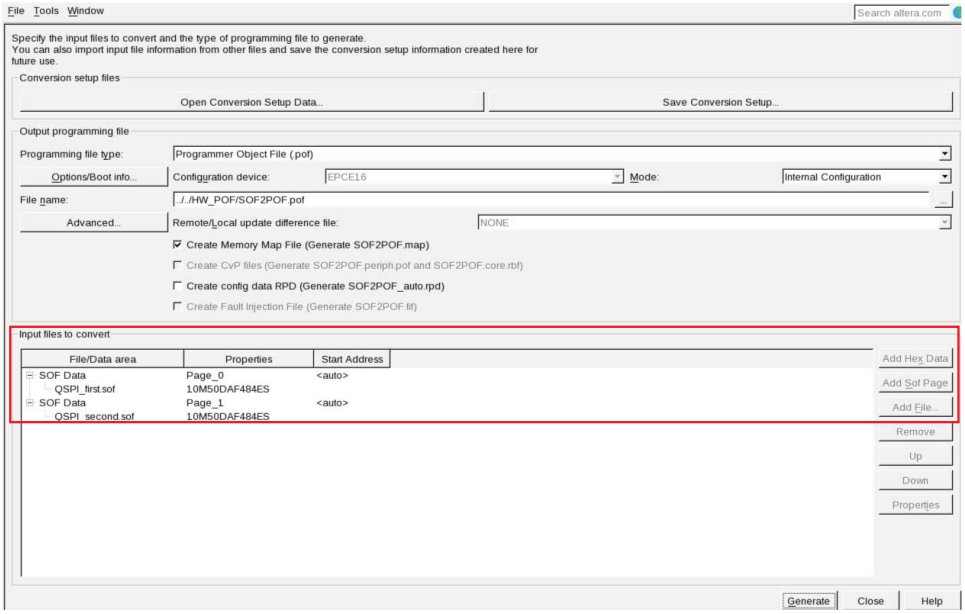
5.2.4.2.3. Programming Files Generation

Note: The following section is only applicable to Intel MAX 10. For other devices, refer to Programming on page 261.

Hardware Programmer Object File (.pof) Generation

1. In Intel Quartus Prime, click on **Convert Programming Files** from the **File** tab.
2. Choose **Programmer Object File (.pof)** as **Programming file type**.
3. Set **Mode** to **Internal Configuration**.
4. Change the **File name** to the desired path and name.
5. Under **Input files to convert**, select **Page_0** and click on **Add File** button on the right.
6. Browse to the first generated `.sof` file and click **OK**.
7. For Dual mode configuration, click on **Add Sof Page** to create additional page for `.sof` file. This creates SOF data **Page_1** automatically.
8. Click **Add File...** and point to the second `.sof` file to add into **Page_1**.
9. Click **Generate** to create the `.pof` file.

Figure 125. SOF to POF Files Conversion Settings



Software Programmer Object File (.pof) Generation

Important: The **quartus.ini** file with `PGMIO_SWAP_HEX_BYTE_DATA=ON` content is required to byteswap the programming file during the POF generation. Please create the quartus.ini file or use the quartus.ini available in the related information and place it under Intel Quartus Prime tool directory or project directory before you proceed.

1. In Intel Quartus Prime, click on **Convert Programming Files** from the **File** tab.
2. Choose **Programmer Object File (.pof)** as **Programming file type**.
3. Set **Mode** to **1-bit Passive Serial**.
4. Set **Configuration device** to **CFI_512Mb**.
5. Change the **File name** to the desired path and name.
6. Remove the SOF **Page_0**.
7. Click on **Add HEX Data**, choose the HEX file generated in **HEX Generation** section. Select **Absolute Addressing** and click **OK**.
8. Repeat Step 7 to add second HEX file if you are using Dual Images configuration mode.
9. Click **Generate** to create the .pof file.

Figure 126. HEX to POF file Conversion Settings for Single Image Configuration Mode

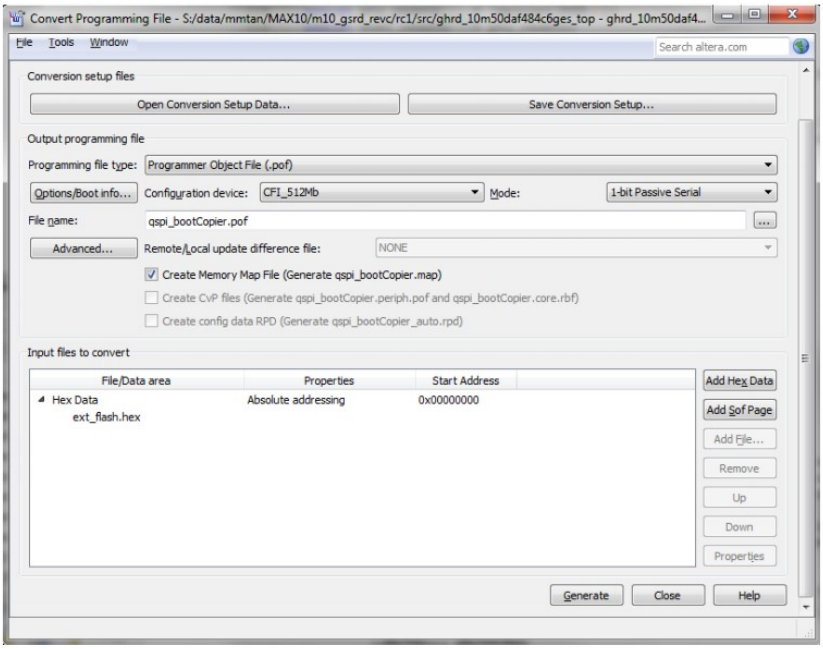
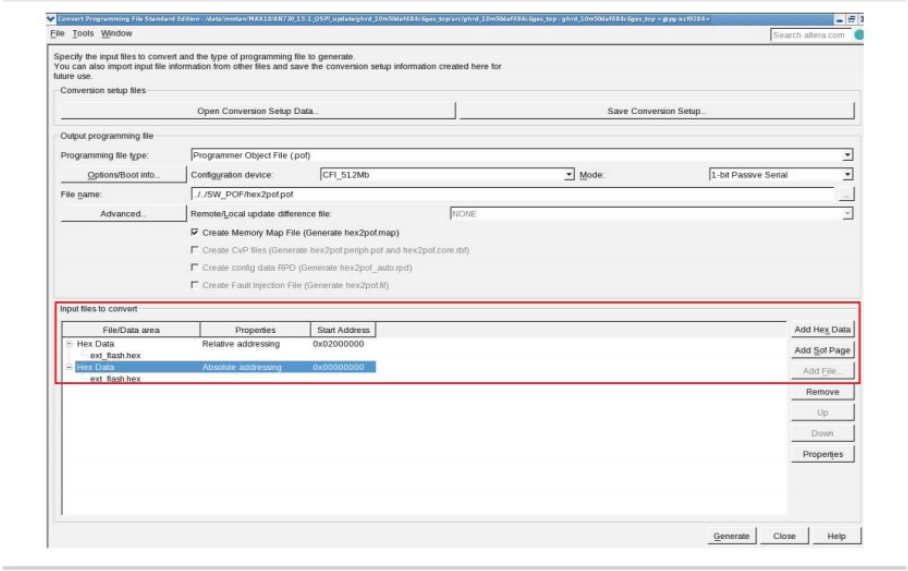


Figure 127. HEX to POF file Conversion Settings for Dual Images Configuration Mode



5.2.4.2.4. QSPI Flash Programming

Note: The following section is only applicable to Intel MAX 10. For other devices, refer to Programming on page 261.

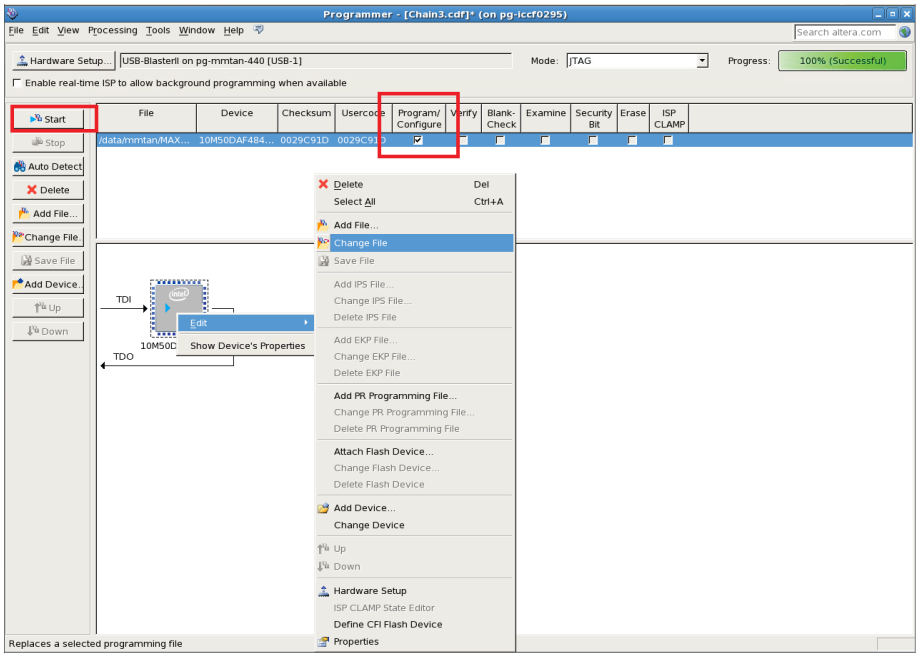
POF File Programming into QSPI Flash

Programming Parallel Flash Loader into Intel MAX 10 Device:

Note: You need to program the parallel flash loader into the Intel MAX 10 device before programming the QSPI flash.

1. Create Parallel Flash Loader for Intel Quartus Prime. Assign QSPI pins based on your design. Compile the project to obtain `max10_qpfl.sof` file.
2. Open Intel Quartus Prime tool (**Tools** -> **Programmer**).
3. Make sure the **Hardware Setup** is set to **USB blaster**.
4. Click on **Auto Detect**, select your Intel MAX 10 FPGA and select **OK**.
5. Right click on the **Intel MAX 10 FPGA** and select **Edit** -> **Change File**. Choose the `max_qpfl.sof` file.
6. Check Intel MAX 10 device under **Program/Configure** and click **Start** to start programming.
7. Click on **Auto Detect** after `max10_qpfl.sof` is successfully programmed. Click **Yes** if you are asked to overwrite the existing settings. A new QSPI flash device will be shown on the screen.

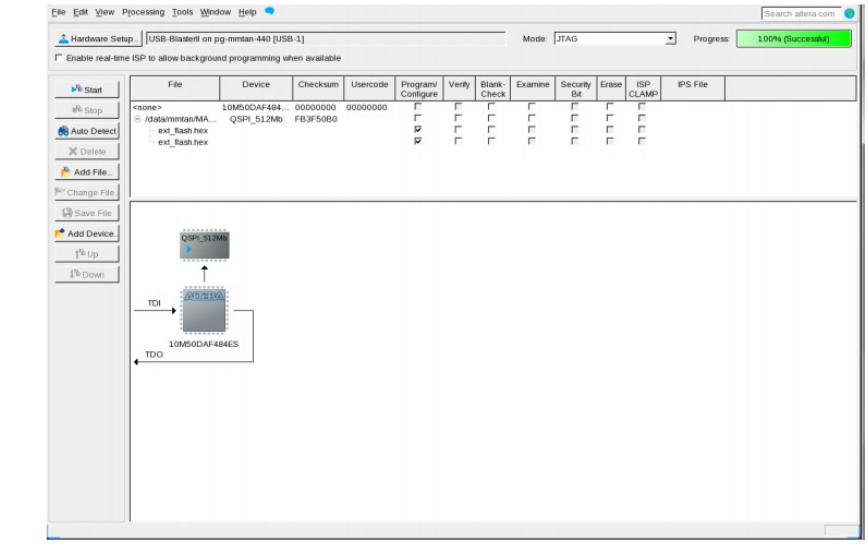
Figure 128. Programming Parallel Flash Loader Programmer Settings



Programming HEX Image into QSPI Flash

1. Right click on the **QSPI device** and select **Edit** -> **Change File**. Choose the generated POF file from *Software POF Generation* section.
2. Check the HEX file under **Program/Configure** column and click **Start** to start programming.

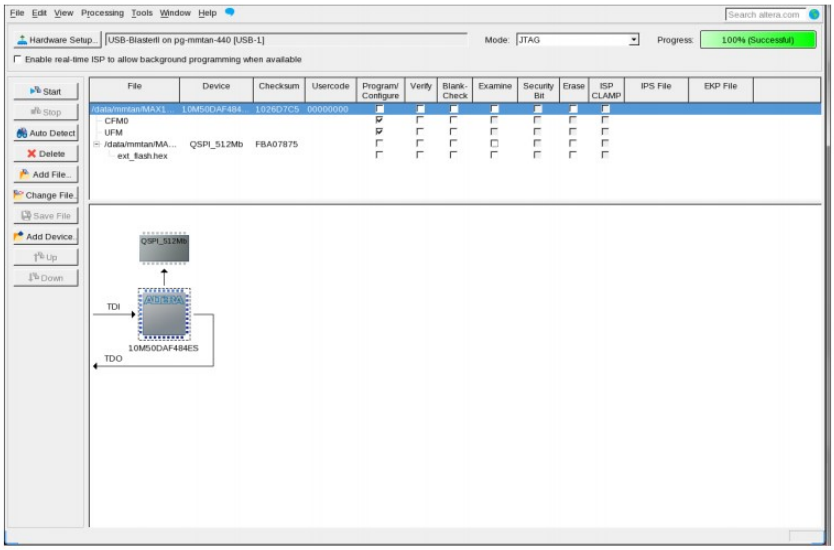
Figure 129. HEX Image Programming into QSPI Flash for Dual Images Configuration Mode



Programming Hardware Design POF File

1. After you successfully programmed HEX data into Quad SPI flash, right click on the **Intel MAX 10 FPGA** and select **Edit** -> **Change File**. Choose the hardware POF file generated from:
   - Intel Quartus Prime project compilation if you are using Single Uncompressed/ Compressed Image configuration mode.
   - Hardware POF Generation section if you are using Dual Compressed Images configuration mode.
2. Check the Intel MAX 10's **CFM0** and **UFM** under **Program/Configure** column and click **Start** to start programming.

Figure 130. POF Image Programming into QSPI Flash



3. Reset your FPGA device using the reset button or power cycle your hardware.

5.2.4.3. Nios II Processor Application Copied from QSPI Flash to RAM Using Boot Copier

Using a boot copier to copy the Nios II application from QSPI flash to RAM is suitable when multiple iterations of application software development and high system performance are required. Nios II processor application copied from QSPI flash to RAM using boot copier but with larger memory space due to QSPI flash (depending on QSPI chip selection). This is an advantage for supporting software applications that require larger program space and high software performance (compared to XIP).

The Nios II SBT tool automatically adds the Nios II processor memcpy-based boot copier to the system when the executable file (`.elf`) is converted to the memory initialization file (`.hex`). The boot copier is located at the base address of the HEX data, followed by the application.

The Nios II processor starts executing the boot copier software upon system reset that copies the application from the QSPI to the on-chip memory or external RAM. Once this process is complete, the Nios II processor transfers the program control over to the application.

Figure 131. Nios II Application Copied from QSPI Flash to RAM using Boot Copier (for Intel MAX 10)

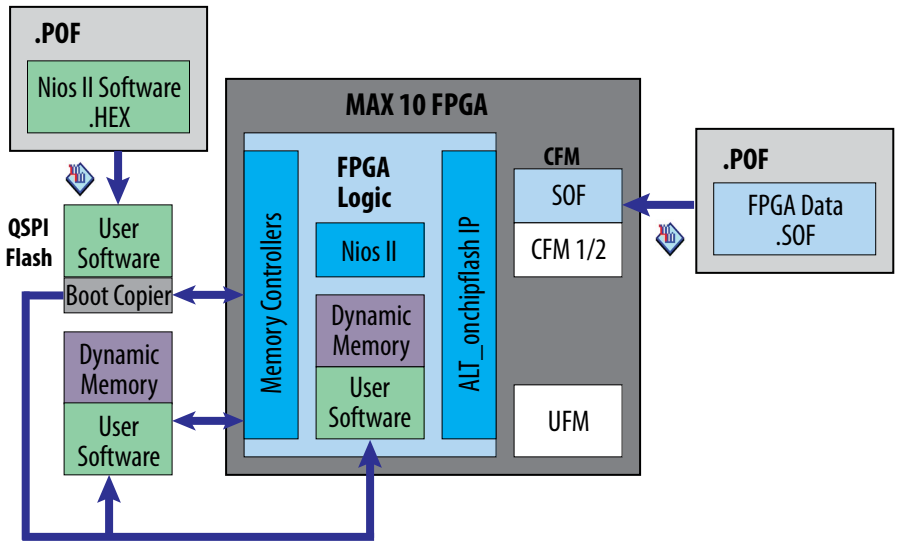


Table 38. RAM and ROM Size Requirement

RAM Size Requirement	ROM Size Requirement
Equivalent to the executable code and dynamic memory size required by user program.	Executable code and boot copier must not exceed the size of the QSPI flash.

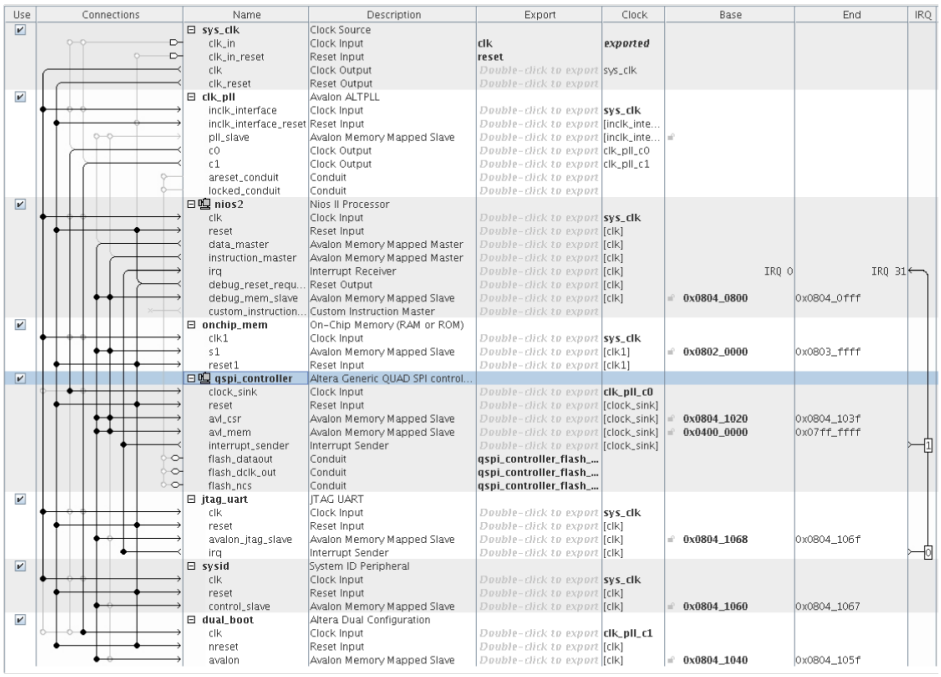
5.2.4.3.1. Hardware Design

The following sections describe a step-by-step method for building a bootable system for a Nios II processor application copied from QSPI flash to RAM using a boot copier.

IP Component Settings

1. Create your Nios II Processor Project using Intel Quartus Prime and Platform Designer.
2. Ensure the Generic Quad SPI Controller IP is added into your Platform Designer system.

Figure 132. Connection for Generic Quad SPI Controller IP



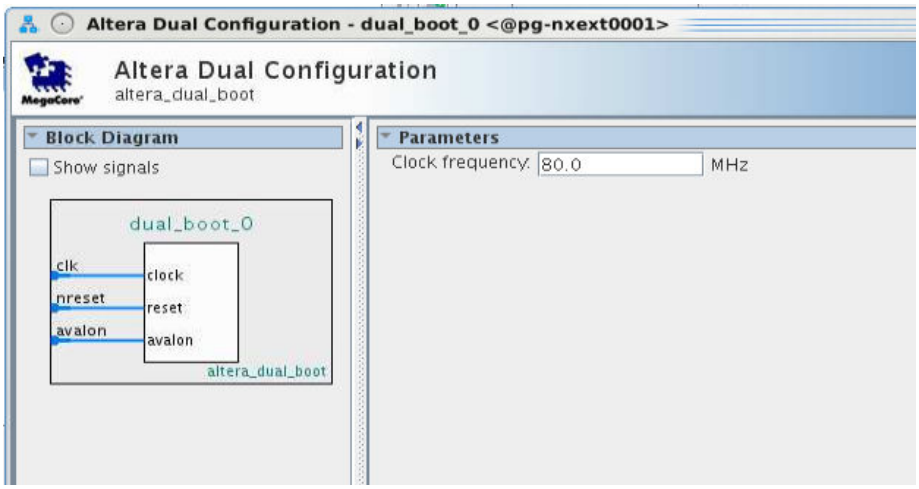
Note: Both Generic QUAD SPI Controller and Generic QUAD SPI Controller II are designed to support serial flash devices from third-party vendors. Generic QUAD SPI Controller II is the newest version and supports higher performance. Intel recommends to use the Generic QUAD SPI Controller II for your new designs.

Important: If you use a serial flash device that is not supported by Generic Quad SPI Controller II, Intel recommends to use Generic Serial Flash Interface IP.

- Generic Serial Flash Interface Intel FPGA IP User Guide

3. For Intel MAX 10 devices, if you want to enable dual images configuration, ensure that Dual Configuration IP is instantiated in your Platform Designer system.

Figure 133. Connection of Dual Compressed Image for Generic Quad SPI Controller IP



4. Open Generic Quad SPI controller parameter editor. Change the Configuration device type to the QSPI flash selection and make sure the I/O mode is set to QUAD.

Figure 134. Generic Quad SPI Controller Parameter Settings



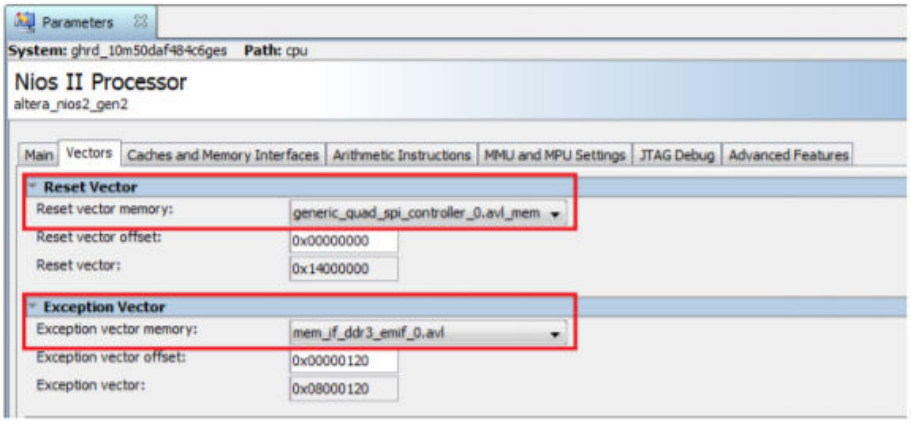
5. Click Generate HDL, the Generation dialog box appears.
6. Specify output file generation options and then click **Generate**.

Reset and Exception Vector Settings for Nios II Using Boot Copier Method

1. In the Nios II Processor parameter editor, set the reset vector memory to QSPI Flash and exception vector memory to OCRAM/ External RAM.

Figure 135. Nios II Parameter Editor Settings

When exception vector is set to OCRAM/ External RAM.



Note: `mem_if_ddr3_emif_0.avl` is the external memory (DDR3) used in this example.

2. For dual configuration images in Intel MAX 10 devices, there are two Nios II application data (HEX file) stored into the QSPI. Reset vector memory offset has to set correctly for the configuration images to call up the correct HEX data.
3. Set Reset vector memory offset of the Nios II Processor in first Platform Designer design to address `0x00000000`.
4. Set Reset vector memory offset of the Nios II Processor in second Platform Designer design to another address to avoid overlapping. For example: Address `0x02000000` which is half of the QSPI memory size (512 Mb).

Figure 136. Reset Vector Offset Setting for First Platform Designer Design

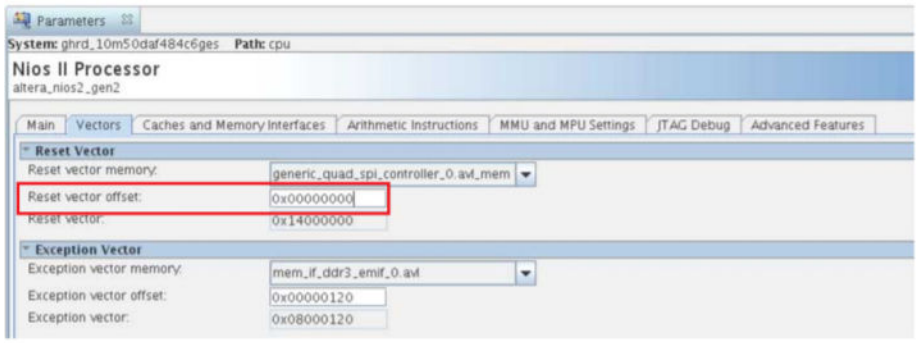
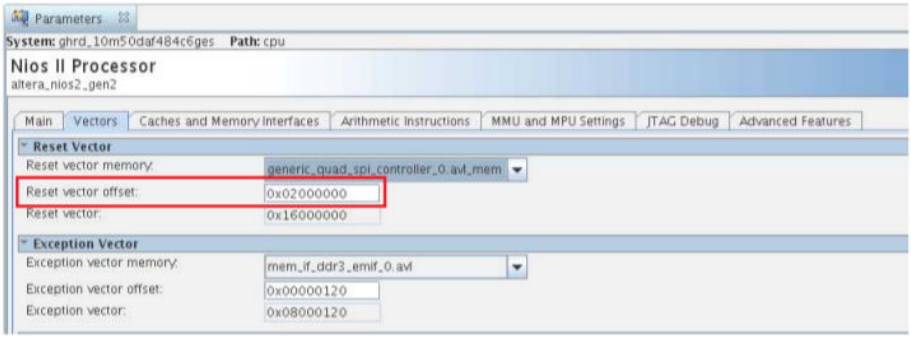


Figure 137. Reset Vector Offset Setting for Second Platform Designer Design

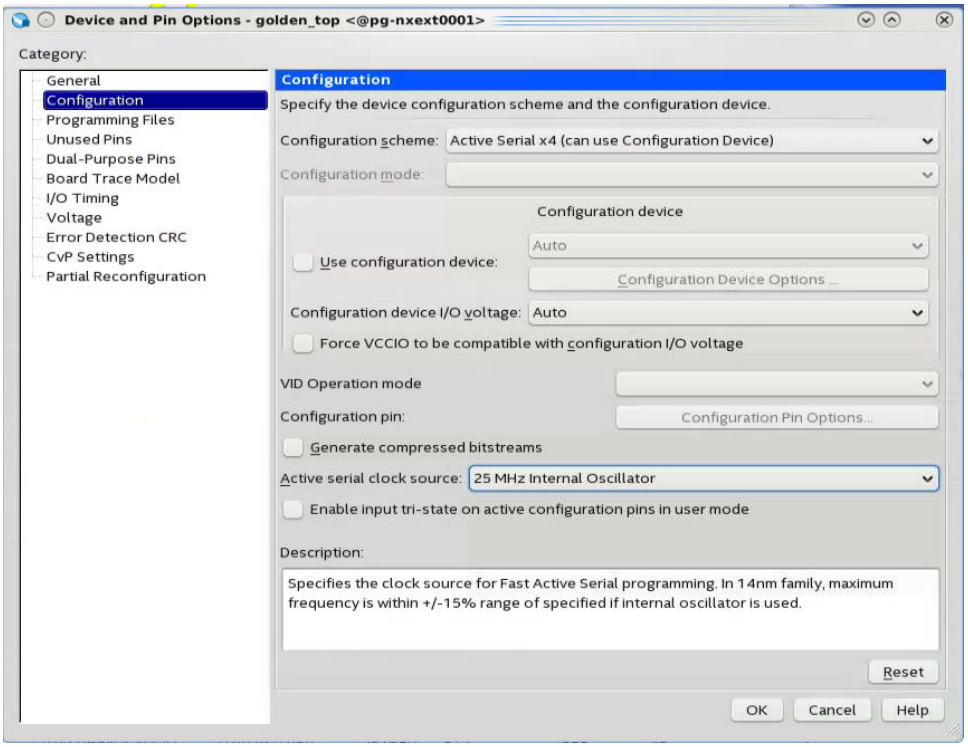


Intel Quartus Prime Software Settings

For non-Max10 devices:

1. In the Intel Quartus Prime software, click on **Assignment ➤ Device ➤ Device and Pin Options ➤ Configuration**.
2. Set **Configuration scheme** to **Active Serial x4 (can use Configuration Device)**.
3. Set the **Active serial clock source** to **25 MHz Internal Oscillator**.

Figure 138. Device and Pin Options

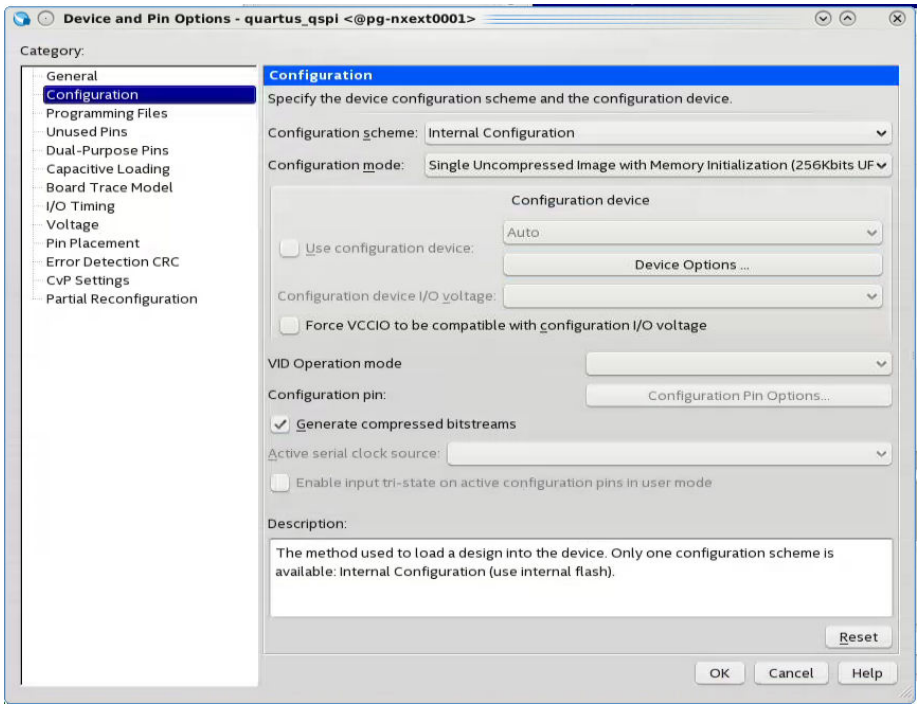


For Intel MAX 10 devices, there will be an option for Single Configuration Image and Dual Configuration Image:

1. In the Intel Quartus Prime software, click on **Assignment ➤ Device ➤ Device and Pin Options ➤ Configuration**. Set **Configuration mode** to **Single Uncompressed Image** or **Single Compressed Image**.

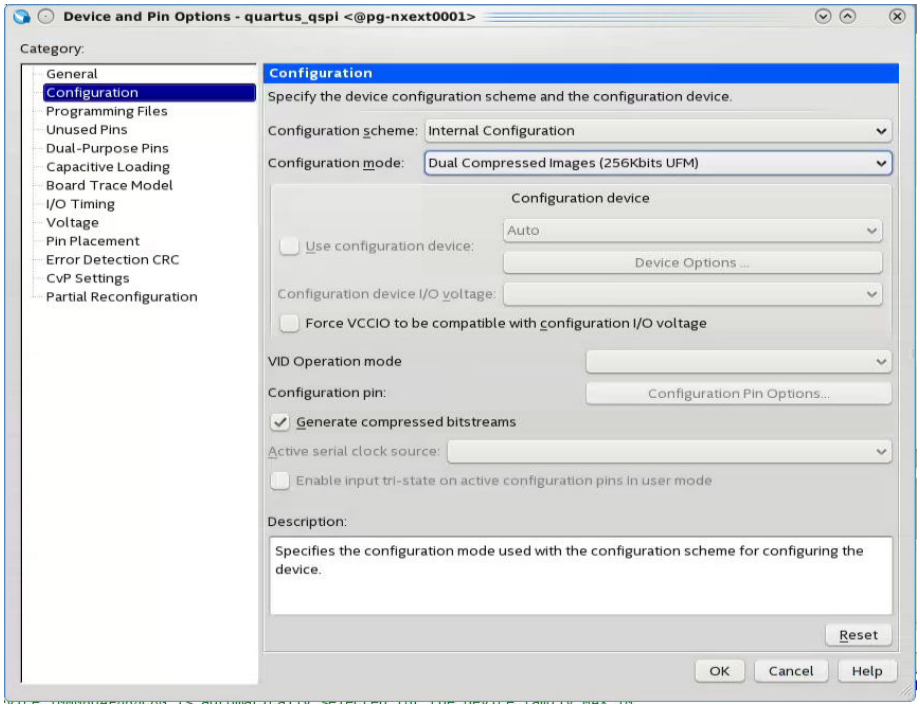
   Note: Intel MAX 10 only supports Single Uncompressed Image with Memory Initialization

Figure 139. Configuration Mode Selection in Intel Quartus Prime Software



2. If you want to set it to Dual Configuration Image, in the Intel Quartus Prime software, click on **Assignment -> Device -> Device and Pin Options -> Configuration**. Set **Configuration mode** to **Dual Compressed Images**.

Figure 140. Dual Configuration Mode Selection in Quartus Prime Software



3. Click **OK** to exit the **Device and Pin Options** window.
4. Click **OK** to exit the **Device** window.
5. Click **Start Compilation** to compile your project.

   Note: SOF to POF file conversion is not required because there is only single hardware image and Nios II application data will be loaded into the QSPI flash separately. You can use the POF generated duringIntel Quartus Prime project compilation to program into the device.

5.2.4.3.2. Application

In the Nios II SBT window, select either:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.

  or

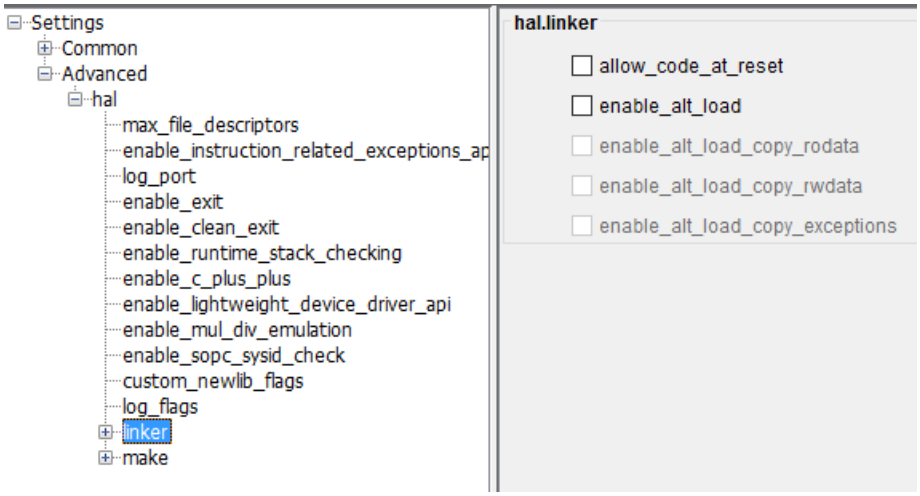
- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

BSP Editor Settings

You must edit the BSP editor settings according to the selected Nios II processor boot options.

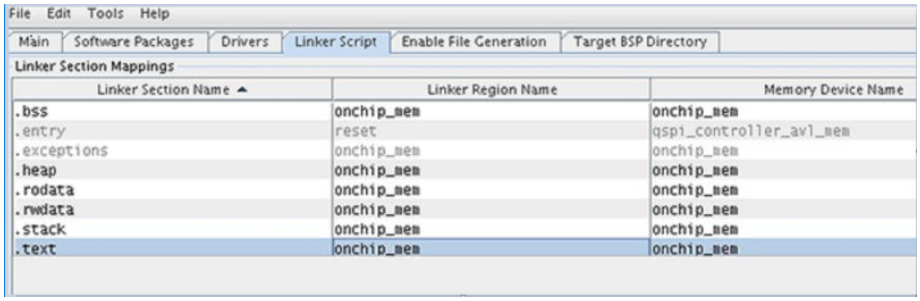
1. In the Nios II SBT tool, right click on your BSP project in the **Project Explorer** window. Select **Nios II ➤ BSP Editor...** to open the **Nios II BSP Editor**.
2. In Nios II BSP Editor, expand into **Advanced.hal.linker** under **Settings**.
3. Leave all the **hal.linker** settings unchecked.

Figure 141. Advanced.hal.linker Settings



4. Click on the **Linker Script** tab in the Nios II BSP Editor.
5. Set all the **Linker Section Name** list to the On-chip Memory (OCRAM) or external RAM.

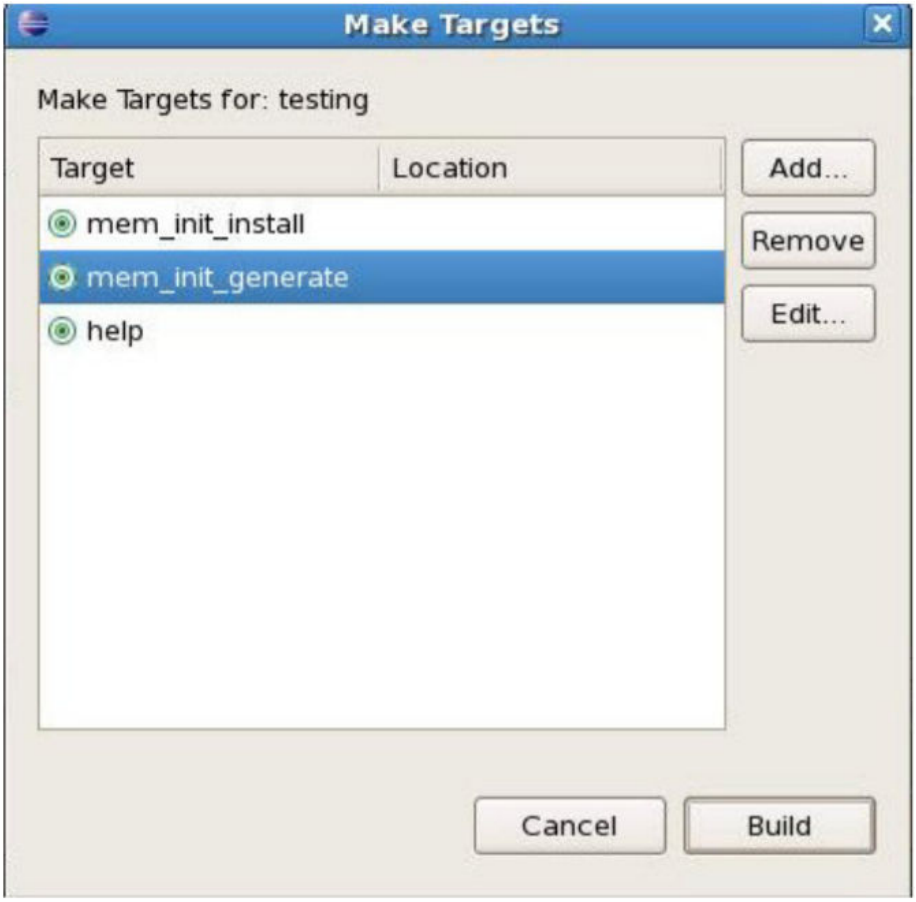
Figure 142. Linker Region Settings



HEX File Generation

1. In the **Nios II SBT tool**, right click on your project in the **Project Explorer** window.
2. Click **Make Targets** -> **Build...**, the **Make Targets** dialog box appears.
3. Select **mem_init_generate**.
4. Click **Build** to generate the HEX file.

Figure 143. Selecting `mem_init_generate` in Make Targets



5. The "mem_init_generate" macro will create two HEX files; `<OCRAM_name>.hex` and `<QSPIFlash_name>.hex`. Select `<QSPIFlash_name>.hex`.

5.2.4.3.3. Programming Files Generation

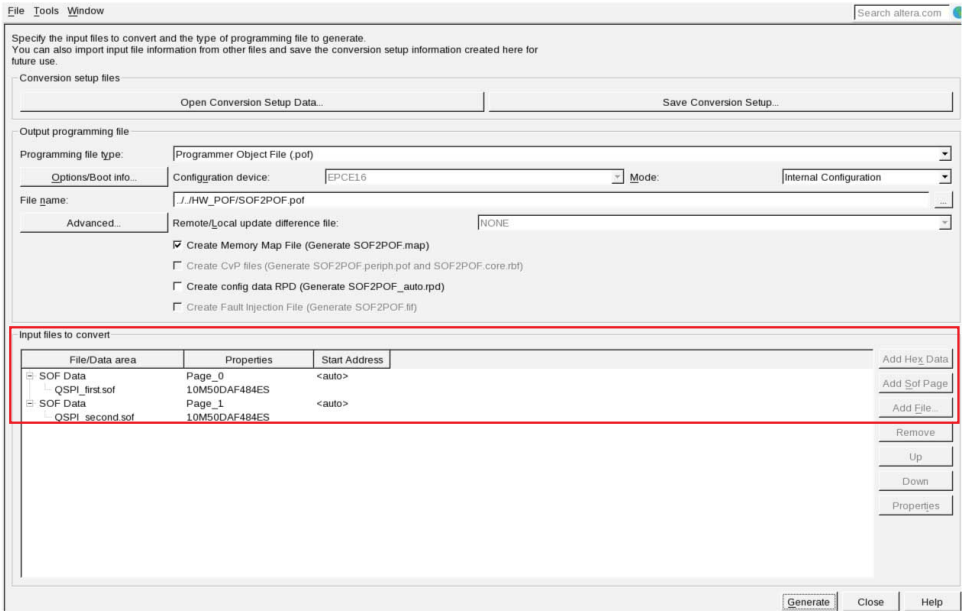
Note: The following section is only applicable to Intel MAX 10. For other devices, refer to Programming on page 270.

Hardware Programmer Object File (.pof) Generation

1. In Intel Quartus Prime, click on **Convert Programming Files** from the **File** tab.
2. Choose **Programmer Object File (.pof)** as **Programming file type**.
3. Set **Mode** to **Internal Configuration**.
4. Change the **File name** to the desired path and name.
5. Under **Input files to convert**, select **Page_0** and click on **Add File** button on the right.
6. Browse to the first generated `.sof` file and click **OK**.

7. For Dual configuration mode, click on **Add Sof Page** to create additional page for `.sof` file. This creates SOF data **Page_1** automatically.
8. Click **Add File...** and point to the second `.sof` file to add into **Page_1**.
9. Click **Generate** to create the `.pof` file.

Figure 144. SOF to POF Files Conversion Settings for Dual Configuration Mode



Software Programmer Object File (.pof) Generation

Important: The **quartus.ini** file with `PGMIO_SWAP_HEX_BYTE_DATA=ON` content is required to byteswap the programming file during the POF generation. Please create the quartus.ini file or use the quartus.ini available in the related information and place it under Intel Quartus Prime tool directory or project directory before you proceed.

1. In Intel Quartus Prime, click on **Convert Programming Files** from the **File** tab.
2. Choose **Programmer Object File (.pof)** as **Programming file type**.
3. Set **Mode** to **1-bit Passive Serial**.
4. Set **Configuration device** to **CFI_512Mb**.
5. Change the **File name** to the desired path and name.
6. Remove the SOF **Page_0**.
7. Click on **Add HEX Data**, choose the HEX file generated in **HEX Generation** section. Select **Absolute Addressing** and click **OK**.
8. Repeat Step 7 to add second HEX file if you are using Dual Images configuration mode.
9. Click **Generate** to create the .pof file.

Figure 145. HEX to POF file Conversion Settings for Single Image Configuration Mode

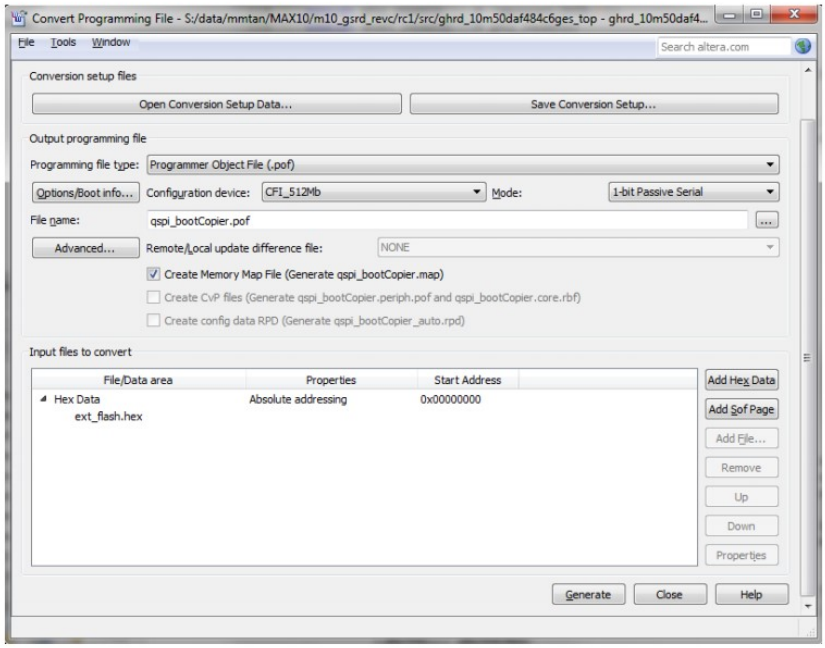
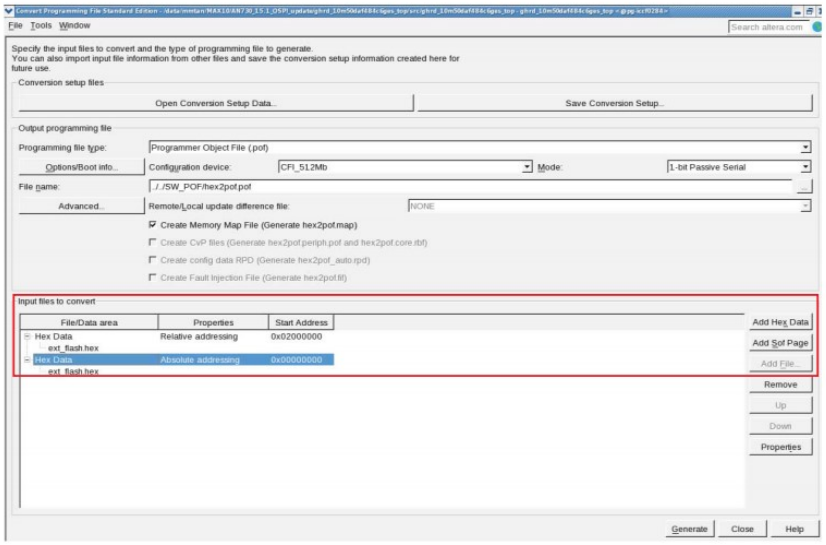


Figure 146. HEX to POF file Conversion Settings for Dual Images Configuration Mode



5.2.4.3.4. QSPI Flash Programming

Note: The following section is only applicable to Intel MAX 10. For other devices, refer to Programming on page 270.

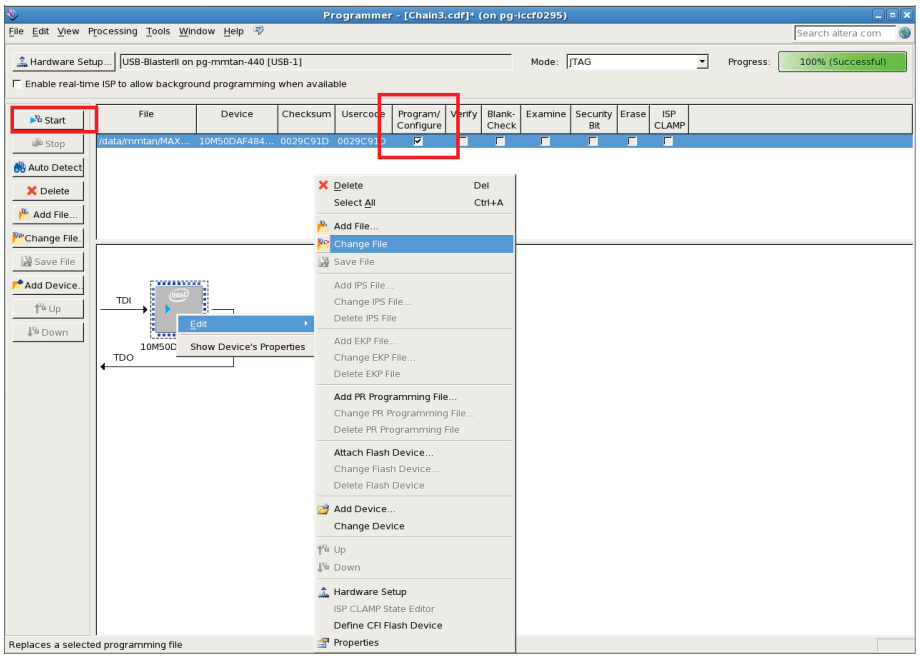
POF File Programming into QSPI Flash

Programming Parallel Flash Loader into Intel MAX 10 Device:

Note: You need to program the parallel flash loader into the Intel MAX 10 device before programming the QSPI flash.

1. Create Parallel Flash Loader for Intel Quartus Prime. Assign QSPI pins based on your design. Compile the project to obtain `max10_qpfl.sof` file.
2. Open Intel Quartus Prime tool (**Tools** -> **Programmer**).
3. Make sure the **Hardware Setup** is set to **USB blaster**.
4. Click on **Auto Detect**, select your Intel MAX 10 FPGA and select **OK**.
5. Right click on the **Intel MAX 10 FPGA** and select **Edit** -> **Change File**. Choose the `max_qpfl.sof` file.
6. Check Intel MAX 10 device under **Program/Configure** and click **Start** to start programming.
7. Click on **Auto Detect** after `max10_qpfl.sof` is successfully programmed. Click **Yes** if you are asked to overwrite the existing settings. A new QSPI flash device will be shown on the screen.

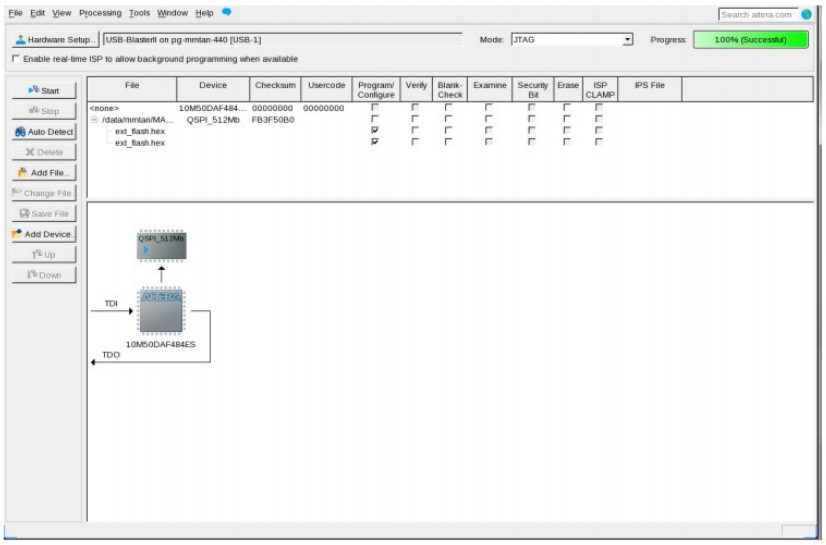
Figure 147. Programming Parallel Flash Loader Programmer Settings



Programming HEX Image into QSPI Flash

1. Right click on the **QSPI device** and select **Edit** -> **Change File**. Choose the generated POF file from *Software POF Generation* section.
2. Check the HEX file under **Program/Configure** column and click **Start** to start programming.

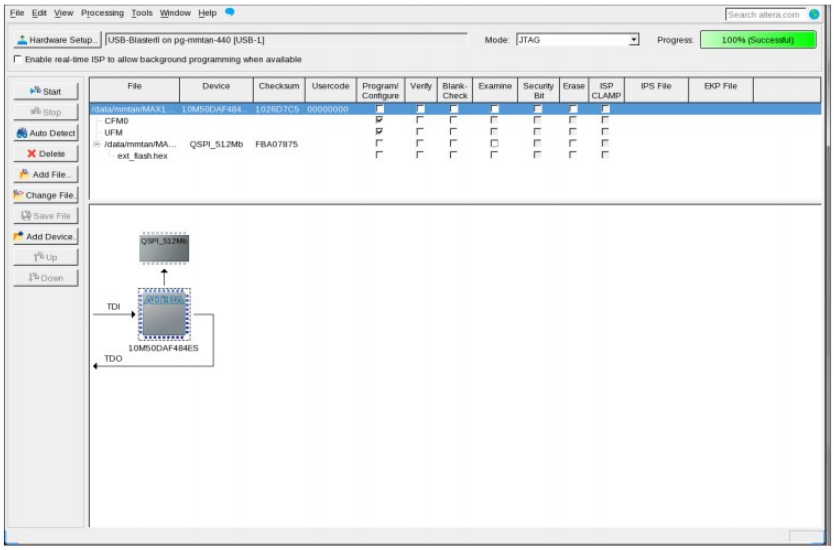
Figure 148. HEX Image Programming into QSPI Flash for Dual Images Configuration Mode



Programming Hardware Design POF File

1. After you successfully programmed HEX data into Quad SPI flash, right click on the **Intel MAX 10 FPGA** and select **Edit** -> **Change File**. Choose the hardware POF file generated from Hardware POF Generation section.
2. Check the Intel MAX 10's **CFM0** and **UFM** under **Program/Configure** column and click **Start** to start programming.

Figure 149. POF Image Programming into QSPI Flash



3. Reset your FPGA device using the reset button or power cycle your hardware.

Note: For non-Max10 device, you can use `.jic` programming flow. Refer to EPCQ programming section.

5.2.5. Nios II Processor Booting from On-Chip Memory (OCRAM)

This section describes the Nios II processor booting and executing software from On-Chip Memory (OCRAM) available in all FPGA devices.

5.2.5.1. Nios II Processor Application Executes in-place from OCRAM

The on-chip memory is initialized during FPGA configuration with data from a Nios II application image. This data is built into the FPGA configuration bitstream. This process eliminates the need for a boot copier, as the Nios II application is already in place at system reset.

Figure 150. Nios II Application XIP from OCRAM in Intel MAX 10 Device

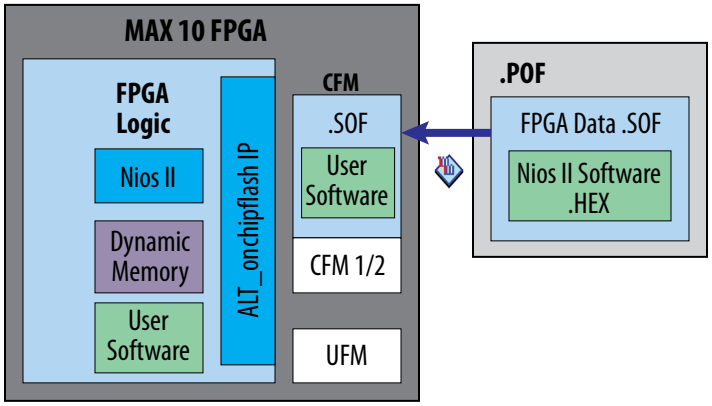


Figure 151. Nios II Application XIP from OCRAM in FPGA Device Configured from Flash Memory

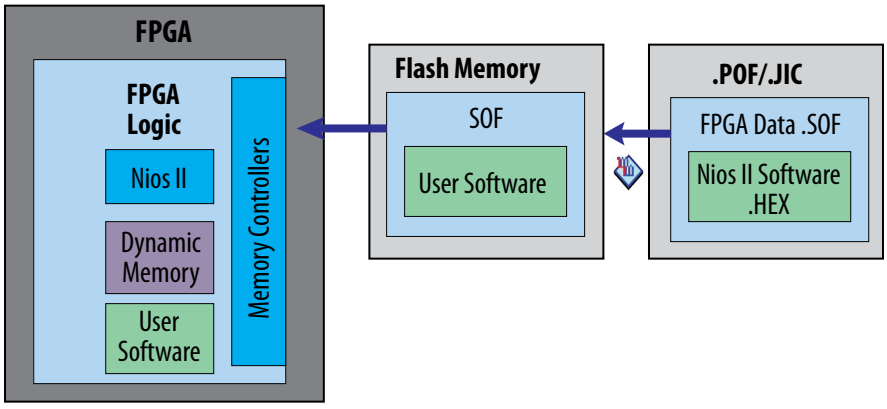
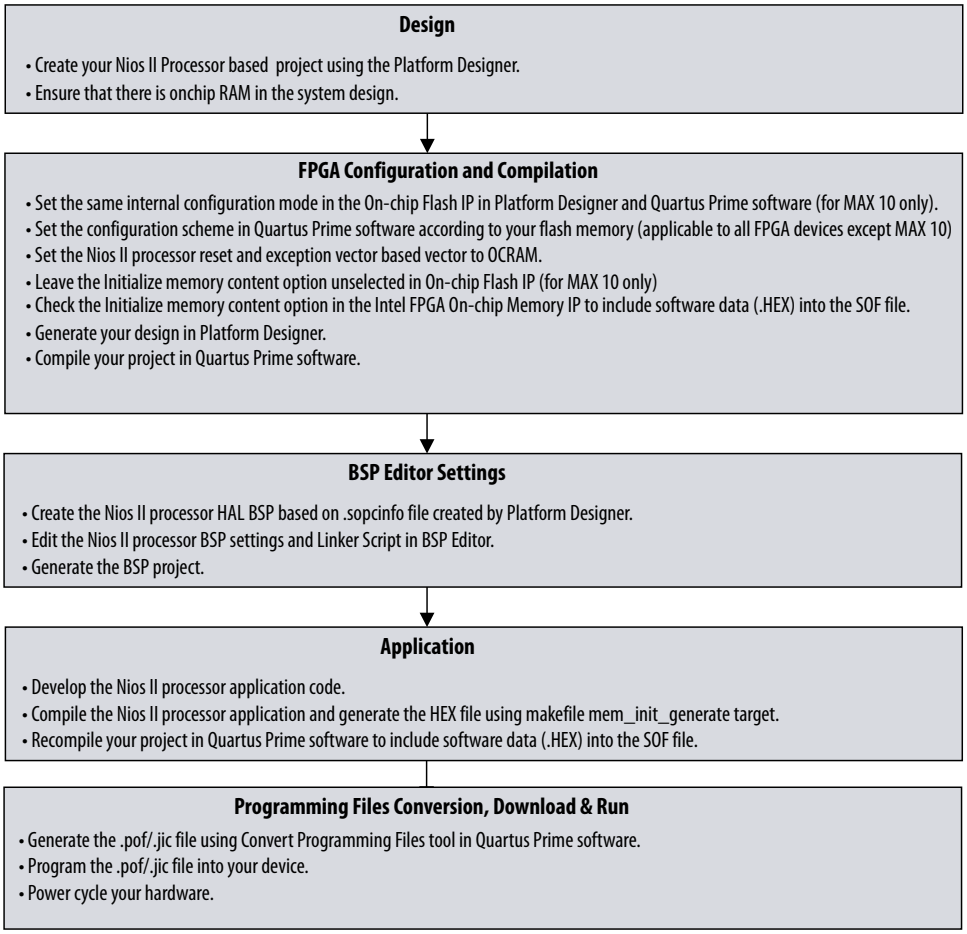


Table 39. RAM and ROM Size Requirement

RAM Size Requirement	ROM Size Requirement
Equivalent to the executable code and dynamic memory size required by user program.	Not applicable for this boot option.

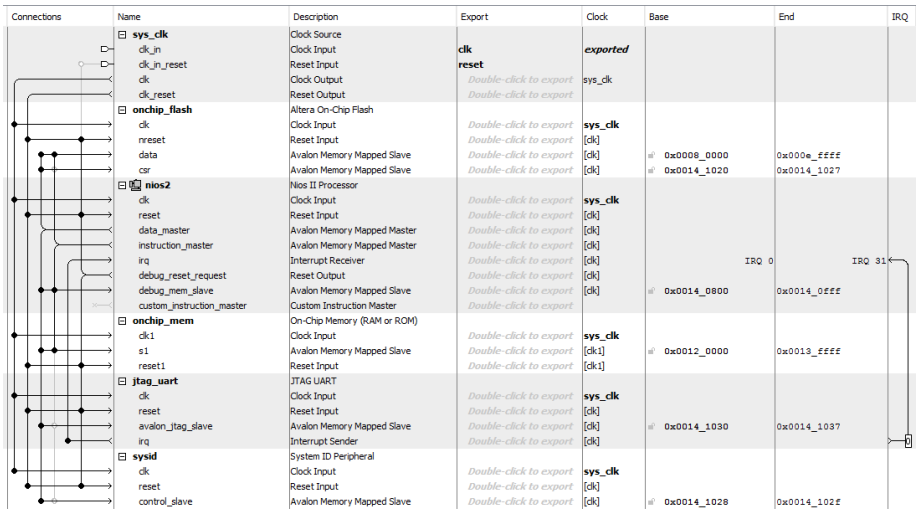
Figure 152. Configuration and Booting Flow



5.2.5.1.1. Hardware Design

1. Create your Intel Quartus Prime and Platform Designer design.
2. Make sure onchip memory (OCRAM) is added into your Platform Designer system.
3. Refer to the diagram below for example IP connections in Platform Designer.

Figure 153. Example IP Connections in Platform Designer for Booting Nios II from On-Chip Memory (OCRAM)

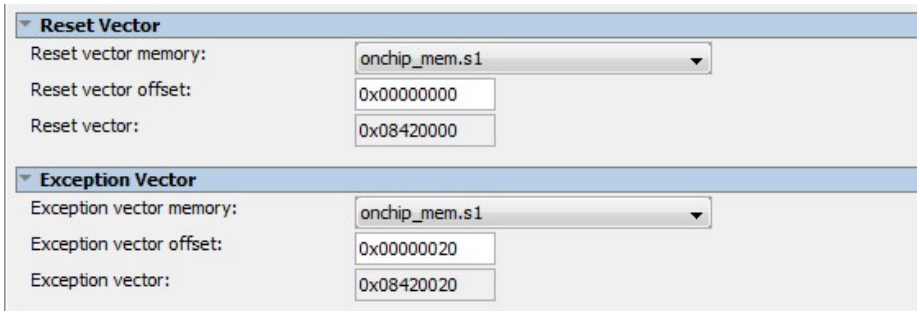


Note: On-Chip Flash IP is applicable to Intel MAX 10 devices only. For other FPGA devices, it should be the memory controller IP that works with your FPGA configuration scheme.

IP Component Settings

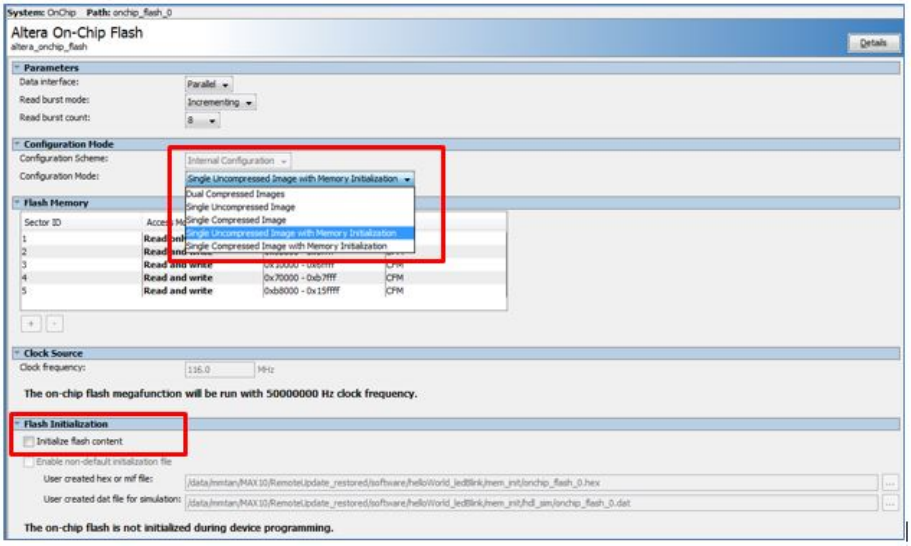
1. In the Nios II Processor parameter editor, set both the **Reset vector memory** and **Exception vector memory** to On-Chip Memory (OCRAM).

Figure 154. Nios II Parameter Editor Setting



2. For Intel MAX 10 devices, in the On-chip Flash IP parameter editor, set the **Configuration Mode** to **Single Uncompressed Image with Memory Initialization** or **Single Compressed Image with Memory Initialization**. Leave the **Initialize flash content** unchecked. This is because the On-chip flash initialization data will not be enabled.

Figure 155. Configuration Mode with Memory Initialization Selection and Initialize Flash Content Setting



3. In the On-chip Memory (RAM or ROM) IP parameter editor, check **Initialize flash content**. If default path is used, add `meminit.qip` generated during "make mem_init_generate" into Intel Quartus Prime project. Make sure the generated HEX naming matches the default naming. If non-default path is selected, check **Enable non-default initialization file** and specify the path of the HEX file (`<on_chip_ram>.hex`).

   Note: The `meminit.qip` stores the location information of the initialization files.

Figure 156. Enable Initialize Memory Content with Default Initialization File in On-Chip Memory Parameter Editor Settings

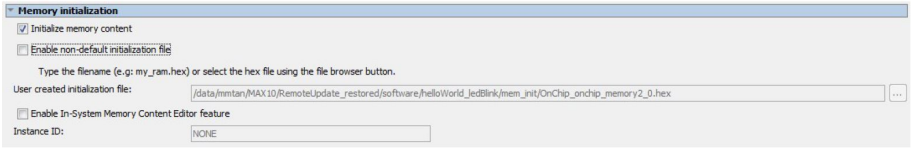


Figure 157. Adding `meminit.qip` File into Intel Quartus Prime

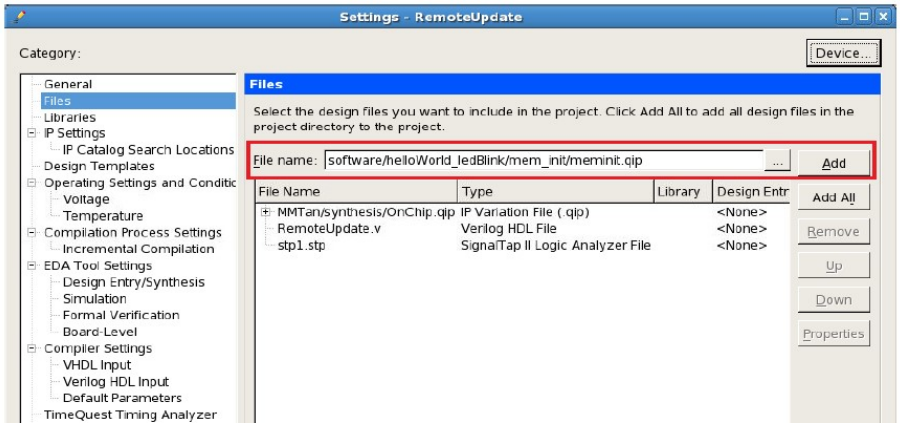
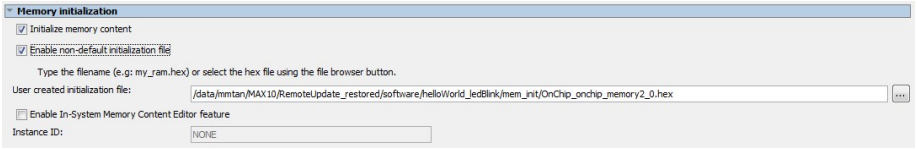


Figure 158. Enable Initialize Memory Content with Non-default Initialization File in On-Chip Memory Parameter Editor Settings



4. Click **Generate HDL**, the **Generation** dialog box appears.
5. Specify output file generation options, and then click **Generate**.

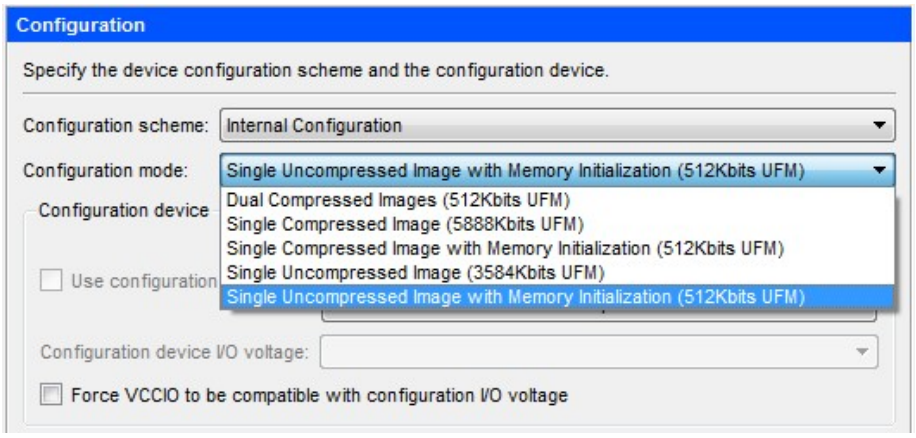
Intel Quartus Prime Software Settings

1. In the Intel Quartus Prime software, click on **Assignment** -> **Device** -> **Device and Pin Options** -> **Configuration**. Set **Configuration mode** to **Single Uncompressed Image with Memory Initialization** or **Single Compressed Image with Memory Initialization**.[18]

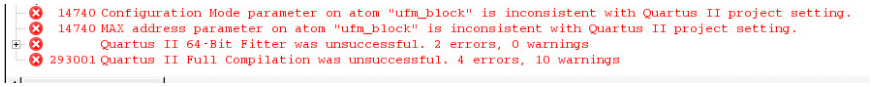
   Note: This setting is applicable to Intel MAX 10 devices only. For other FPGA devices, set the **Configuration mode** according to your FPGA configuration scheme.

[18] The size of UFM sector will vary according to your device selection.

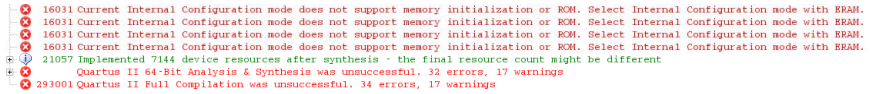
Figure 159. Configuration Modes with Memory Initialization Selection in Intel Quartus Prime



Note:
- If the configuration mode setting in Intel Quartus Prime software and Platform Designer parameter editor is different, the Intel Quartus Prime project compilation will fail with the following error message.



- If configuration mode without memory initialization is selected with **Initialize flash content** checked in On-chip Memory IP, the Intel Quartus Prime project compilation will fail with the following error message:



2. Click **OK** to exit the **Device and Pin Options window**.
3. Click **OK** to exit the **Device** window.
4. Click **Start Compilation** to compile your project and generate the `.sof` file.

5.2.5.1.2. Application

In the Nios II SBT window, select either:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.

  or

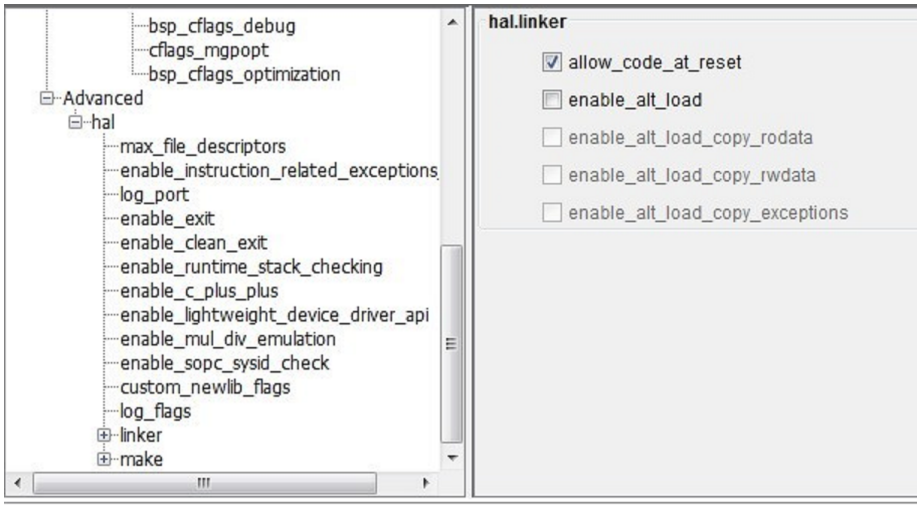
- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

BSP Editor Settings

You must edit the BSP editor settings according to the selected Nios II processor boot options.

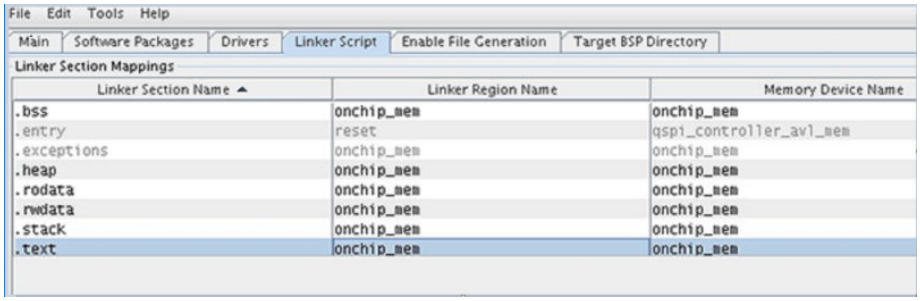
1. In the Nios II SBT tool, right click on your BSP project in the **Project Explorer** window. Select **Nios II ➤ BSP Editor...** to open the **Nios II BSP Editor**.
2. In Nios II BSP Editor, expand into **Advanced.hal.linker** under **Settings**.
3. Enable **allow_code_at_reset** and leave others unchecked to make sure the application starts at address 0x00.

Figure 160. Advanced.hal.linker Settings



4. Click on the **Linker Script** tab in the Nios II BSP Editor.
5. Set all the **Linker Section Regions** to the On-chip Memory (OCRAM).

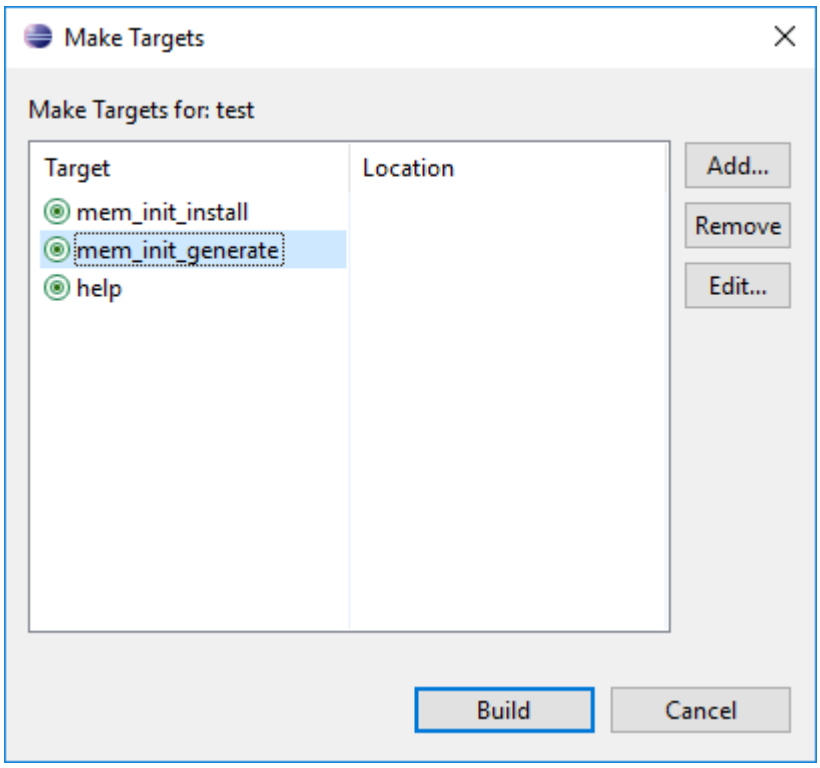
Figure 161. Linker Region Settings



HEX File Generation

1. In the Nios II SBT tool, right click on your project in the Project Explorer window.
2. Click **Make Targets** -> **Build...**, the **Make Targets** dialog box appears. You can also press shift + F9 to trigger the **Make Target** dialog box.
3. Select **mem_init_generate**.
4. Click **Build** to generate the HEX file.

Figure 162. Selecting `mem_init_generate` in Make Targets



5. The "mem_init_generate" macro will create <on_chip_ram.hex>.

   Note:
   - The `mem_init_generate` target also generates a Intel Quartus Prime IP file (`meminit.qip`). Intel Quartus Prime software will refer to the `meminit.qip` for the location of the initialization files.
   - All these files can be found under the "`<project_folder>/software/<application_name>/mem_init`" folder.

6. Recompile your project in Intel Quartus Prime software to include the software data (.HEX) into the SOF file.

5.2.5.1.3. Programming

The Nios II application data is built into the FPGA configuration bitstream. Based on your FPGA configuration scheme, program your FPGA device with the programming file containing .sof file and your Nios II application will get running once the FPGA configured successfully.

5.2.6. Nios II Processor Booting from CFI Flash

This section describes the Nios II processor booting and executing software from CFI flash. The CFI flash is commonly used with Fast Passive Parallel (FPP) configuration scheme. You can use the Avalon Tri-State Conduit components in Intel Quartus Prime software to access the CFI flash.

The MAX series CPLD supports Parallel Flash Loader (PFL) IP to program CFI flash memory devices through JTAG interface and provides the logic to control FPP configuration from the flash memory device to the FPGA device.

Related Information

Parallel Flash Loader IP Core User Guide

5.2.6.1. Description of the Avalon Tri-State Conduit Components

The Avalon Tri-State Conduit components available in the Platform Designer allow you to create on-chip controllers that connect to off-chip devices such as CFI flash.

- The **Generic Tri-State Conduit Controller** includes parameters that you can specify to control the connected off-chip flash device.
- The **Tri-State Conduit Pin Sharer** arbitrates between multiple connected tri-state controllers. It drives signals from the selected controller to the Tri-State Conduit Bridge.
- The **Tri-State Conduit Bridge** converts an on-chip encoding of tri-state signals into true bidirectional signals on the PCB.

Related Information

Avalon Tri-State Conduit Components User Guide

5.2.6.2. Nios II Processor Application Execute-In-Place from CFI Flash

This option is suitable for Nios II processor applications which require limited on-chip memory usage. The `alt_load()` function operates as a mini boot copier which initializes and copies the data sections (**.rodata, .rwdata, or .exceptions**) from CFI flash to RAM based on the BSP settings. The code section (**.text**) which is a read only section, remains in the CFI flash memory region. This setup offers faster boot time because it does not need to copy code to RAM.

The Nios II processor reset vector points to the CFI flash to allow code execution after the system reset. If you are debugging the application using the source-level debugger, you must use a hardware breakpoint because the CFI cannot efficiently support random memory access.

Figure 163. Nios II Application XIP from CFI Flash

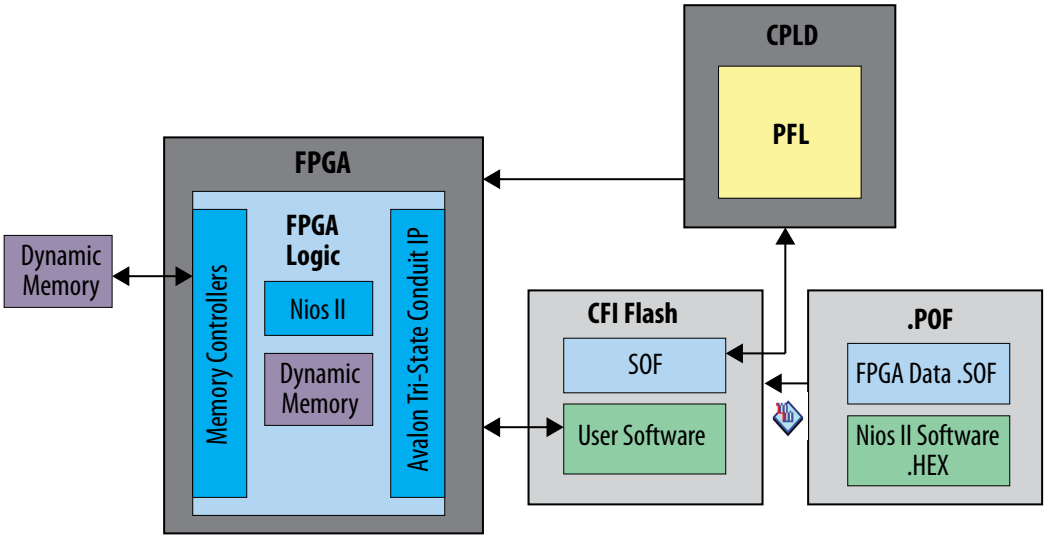
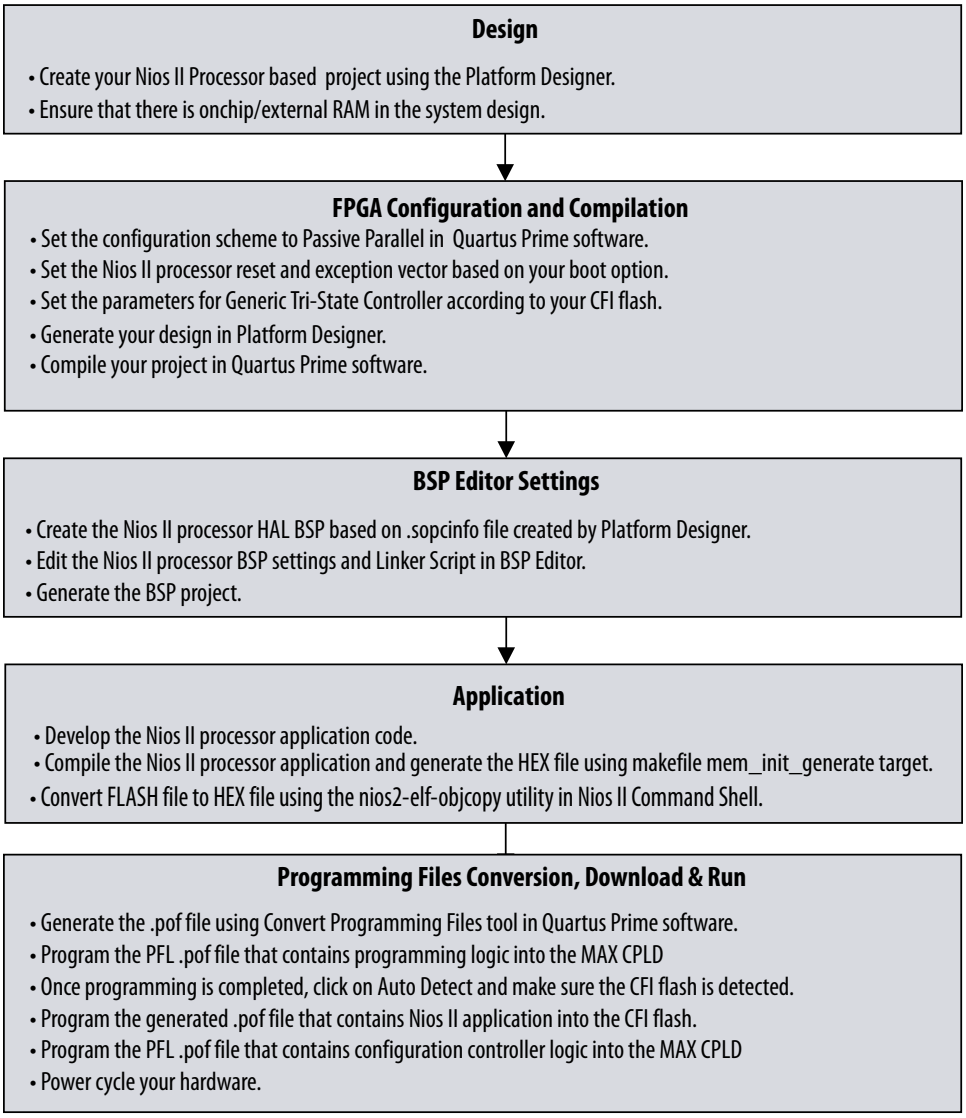


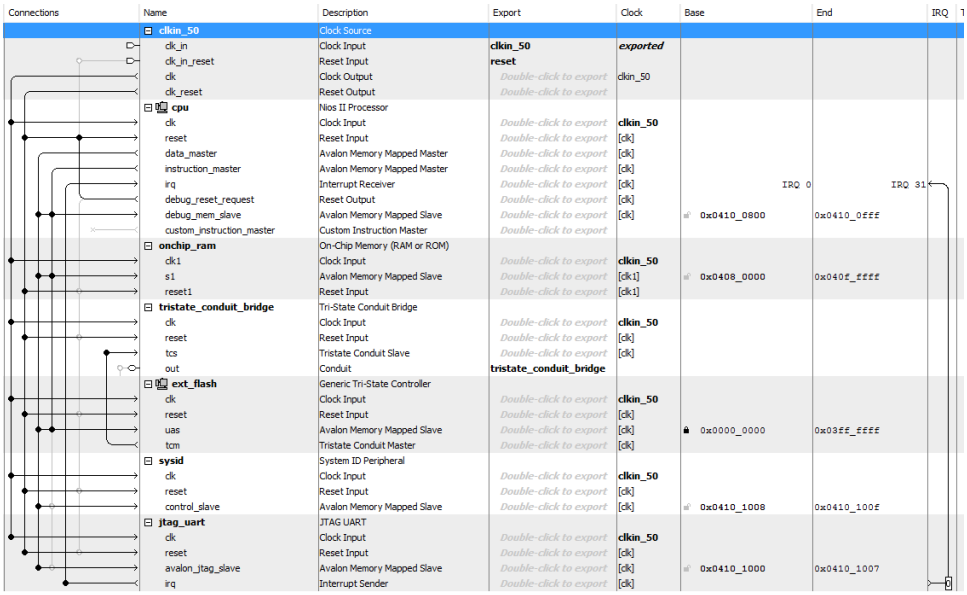
Figure 164. Design, Configuration and Booting Flow



5.2.6.2.1. Hardware Design

1. Create your Intel Quartus Prime project and Platform Designer design.
2. Make sure OCRAM or External RAM is added into your Platform Designer system.
3. Refer to diagram below for example IP connections in Platform Designer.

Figure 165. Example IP Connections in Platform Designer for Booting Nios II from CFI Flash



IP Component Settings

1. In the Nios II processor parameter editor, set the **Reset vector memory** to CFI flash and **Exception vector memory** to OCRAM / External RAM or CFI flash according to your design preference.

   Note: Setting the exception vector memory to OCRAM / External RAM is recommended to make the interrupt processing faster.

   a. If exception vector memory is set to CFI Flash, the minimum exception vector offset that you can set is 0x20.

      Note: When executing-in-place, the Nios II processor boots and runs directly from CFI flash, without copying any code at boot time. Because the Nios II begins executing at the reset address in this case, exception vectors must be located at a nonzero exception vector offset to allow for 32 bytes instructions between the reset vector and the base of the exception vectors.

   b. The **Reset vector** must be the base address of your application. In this example, it is 0x01bc0000.

      Note: Your ***.sof** image size influences your reset vector offset configuration. The reset vector offset is the start address of the **.hex** file in CFI flash and it must point to a location after the ***.sof** image. You can determine the minimum reset vector offset by using the following equation:

```
minimum reset vector offset = (.sof image start address + .sof image
size) in HEX
```

Figure 166. Nios II Parameter Editor Settings with Exception Vector Set to OCRAM / External RAM

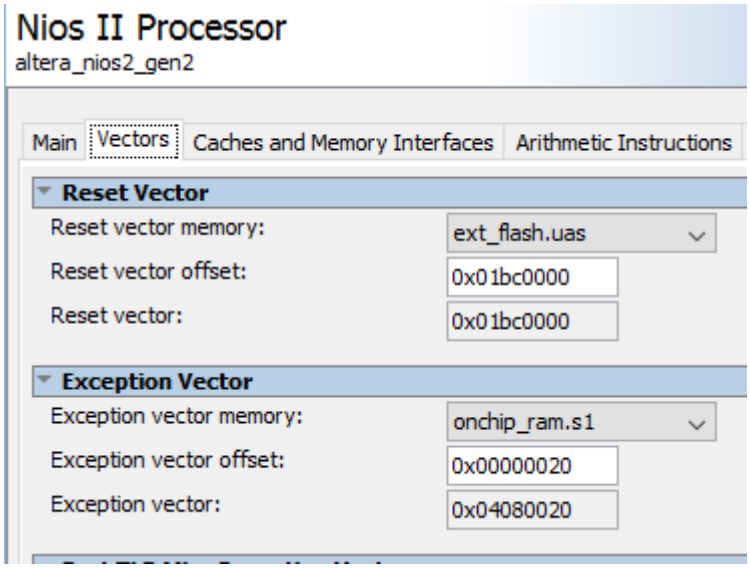
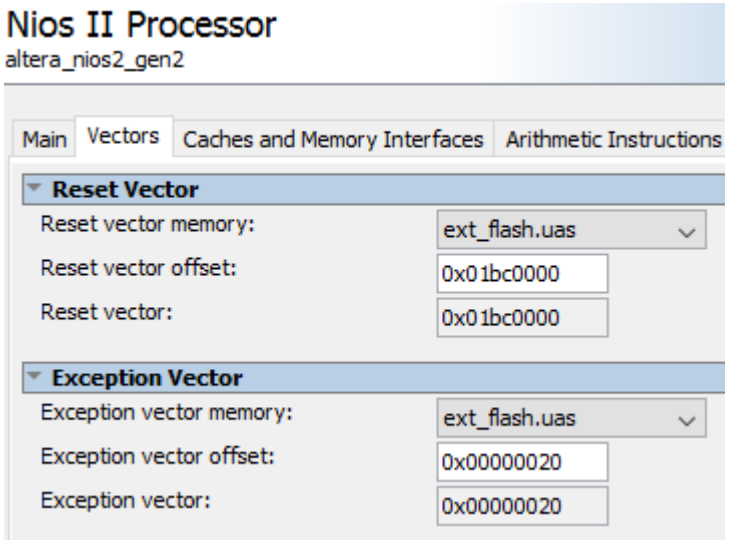


Figure 167. Nios II Parameter Editor Settings with Exception Vector Set to CFI Flash



2. In Generic Tri-State Controller IP parameter editor, select one of the preset configurations in **Library** to automatically assign the parameter values.

Figure 168. Generic Tri-State Controller Preset Configuration



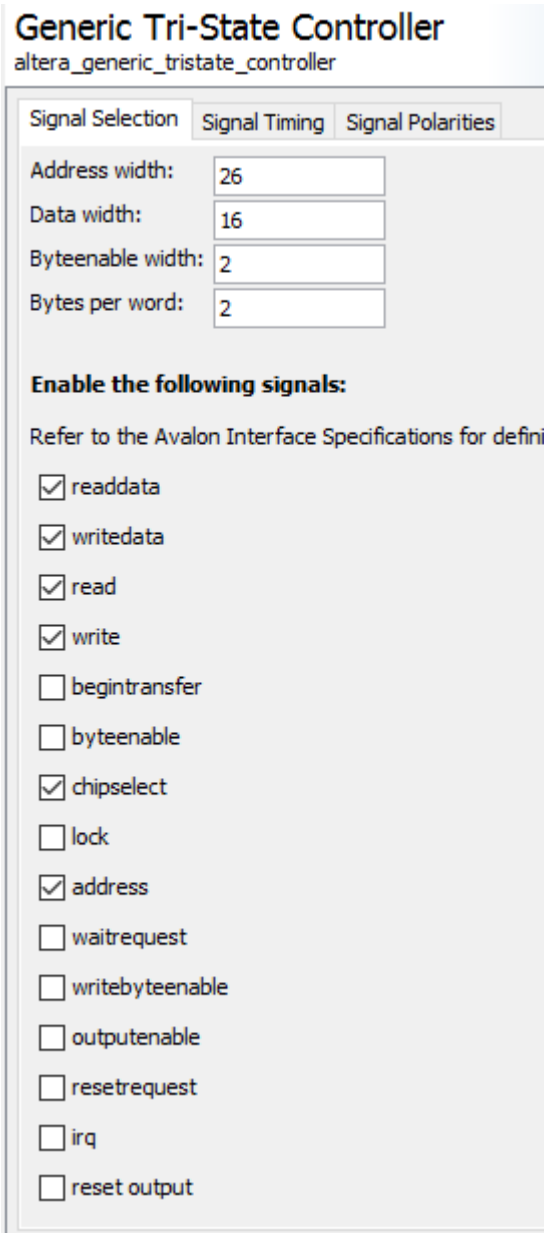
If your flash device is not listed in the **Library**, you can use the parameter editor to specify the required settings.

Note: The appropriate values for these parameters are typically listed in the device's datasheet.

On top of the device specific parameters, ensure the following settings are correct in place:

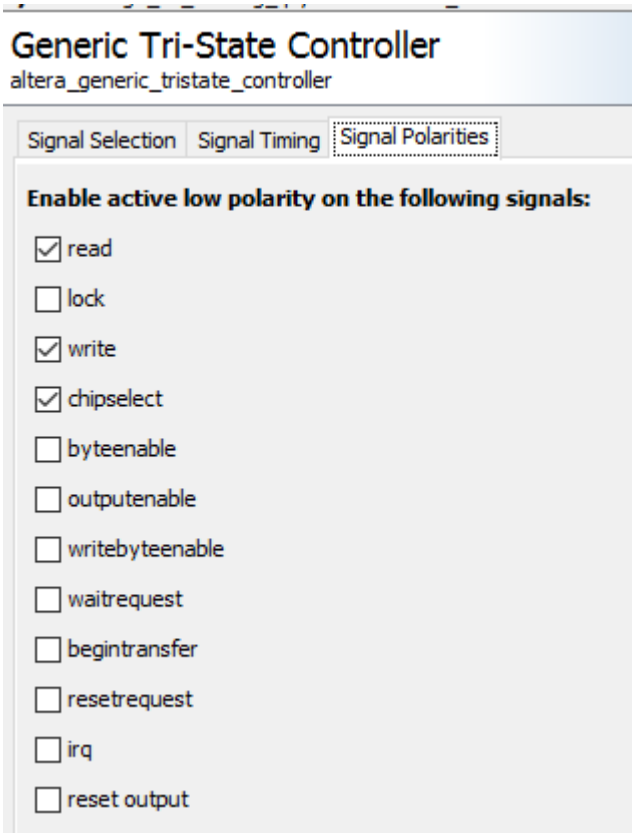
a. In **Signal Selection** tab, enable the following signals:
   - `readdata`
   - `writedata`
   - `read`
   - `write`
   - `chipselect`
   - `address`

Figure 169. Signal Selection



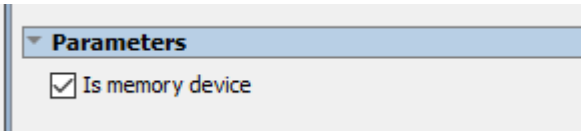
b. In **Signal Polarities** tab, enable the following signals:
   - `read`
   - `write`
   - `chipselect`

Figure 170. Signal Polarities



c. In **Parameters** tab, select **Is memory device**.

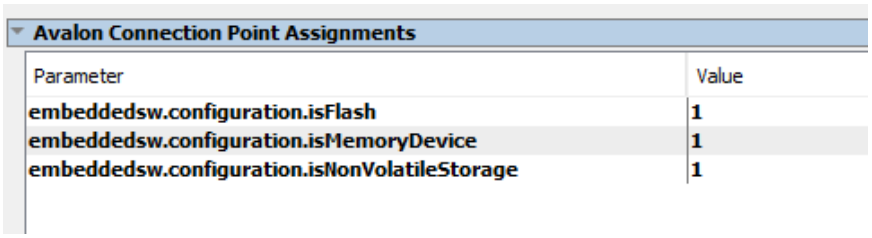
Figure 171. Parameters Window



d. In **Avalon Connection Point Assignments** tab, set the following parameters to a value of 1:

- **embeddedsw.configuration.isFlash**
- **embeddedsw.configuration.isMemoryDevice**
- **embeddedsw.configuration.isNonVolatileStorage**

Figure 172. Avalon Connection Point Assignments

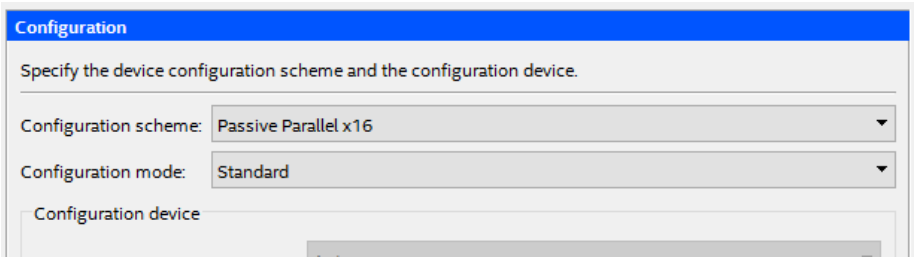


3. Click **Generate HDL**. The **Generation** dialog box appears.
4. Specify output file generation options, and then click **Generate**.

Intel Quartus Prime Software Settings

1. In the Intel Quartus Prime software, click on **Assignments > Device > Device and Pin Options > Configuration**. Set **Configuration scheme** to **Passive Parallel x8/x16/x32**(19)

Figure 173. Configuration Scheme Selection



2. Click **OK** to exit the **Device and Pins Options** window.
3. Click **OK** to exit the **Device** window.
4. Click on **Processing > Start Compilation** to compile your project and generate the .sof file.

5.2.6.2.2. Application

In the Nios II SBT window, select:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.
- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

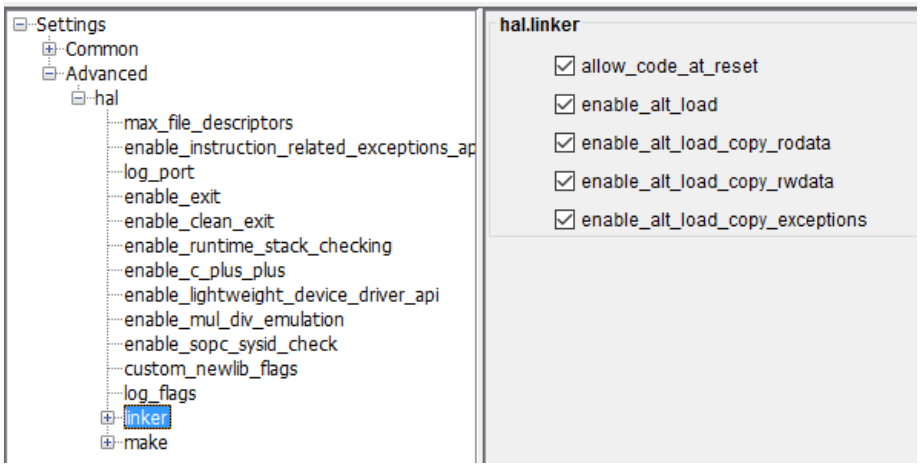
BSP Editor Settings

You must edit the BSP settings according to the selected Nios II processor boot options.

(19) Set data width according to supported scheme.

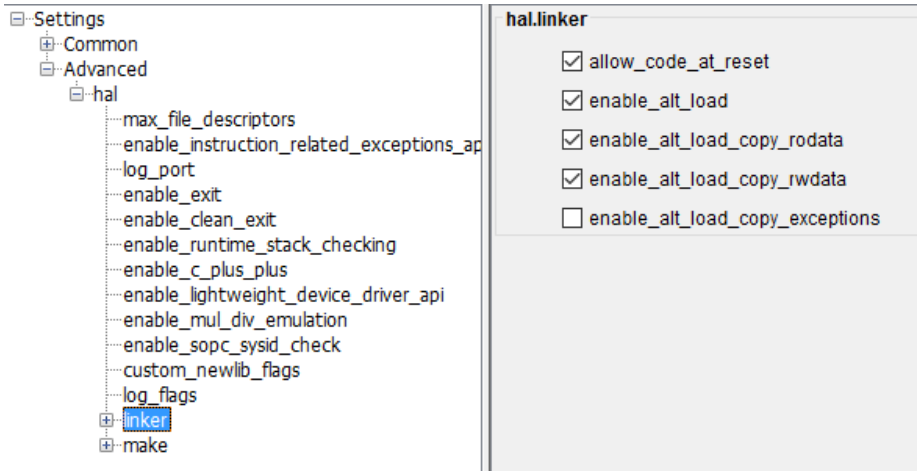
1. In the Nios II SBT tool, right click on your BSP project in the **Project Explorer** window. Select **Nios II > BSP Editor...** to open the Nios II BSP Editor.
2. In Nios II BSP Editor, expand into **Advanced.hal.linker** under **Settings** in **Main** tab.
3. If the exception vector memory is set to OCRAM or External RAM, then enable the following:
   - **allow_code_at_reset**
   - **enable_alt_load**
   - **enable_alt_load_copy_rodata**
   - **enable_alt_load_copy_rwdata**
   - **enable_alt_load_copy_exceptions**

Figure 174. Advanced.hal.linker Settings (Exception vector memory Set to OCRAM or External RAM)



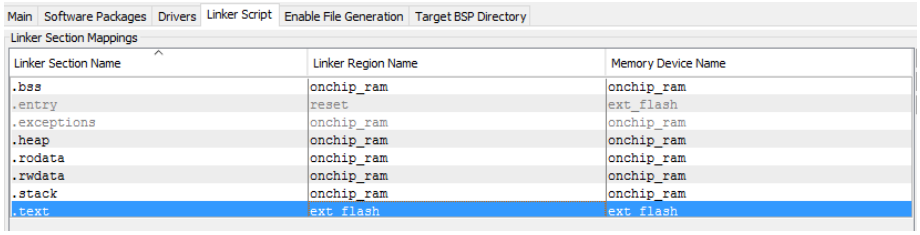
4. If the exception vector memory is set to CFI flash, then enable the following:
   - **allow_code_at_reset**
   - **enable_alt_load**
   - **enable_alt_load_copy_rodata**
   - **enable_alt_load_copy_rwdata**

Figure 175. Advanced.hal.linker Settings (Exception vector memory Set to CFI Flash)



5. Click on the **Linker Script** tab in the Nios II BSP Editor.
6. Set the **.text** region in the **Linker Section Name** to the CFI Flash in the **Linker Region Name**. Set the rest of the regions in the **Linker Section Name** list to the OCRAM or External RAM.

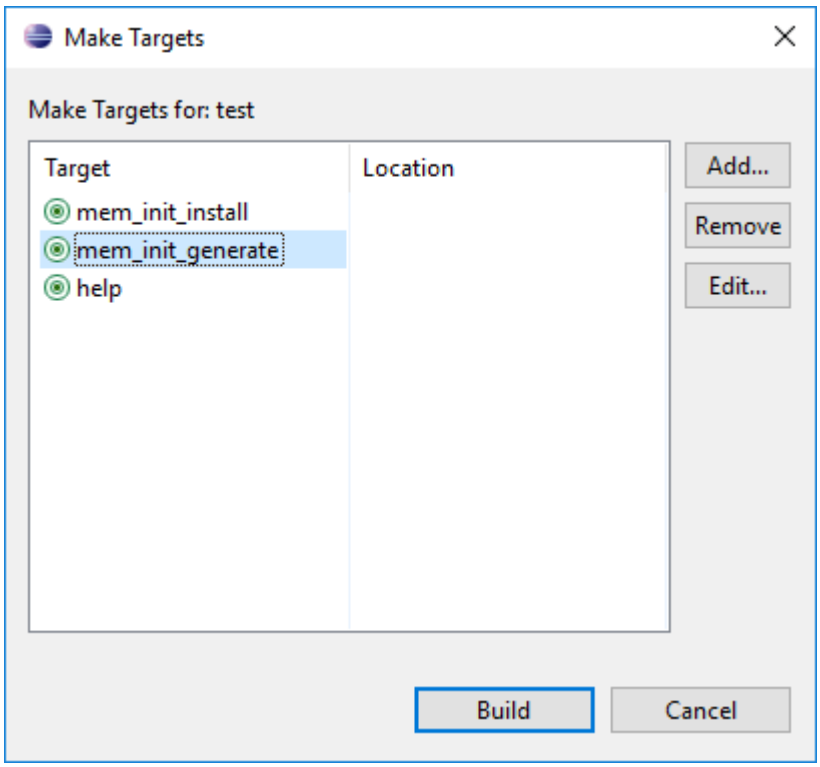
Figure 176. Linker Region Settings



HEX File Generation

1. After creating the Nios II application, right click on the project in **Project Explorer** and select **Build Project**. An ***.elf** file is created under the project folder.
2. Right click again on your project and click on **Make Targets** > **Build...**, the **Make Targets** dialog box appears. You can also press **shift + F9** to trigger the **Make Targets** dialog box.
3. Select **mem_init_generate**.
4. Click **Build**. A ***.flash** file is created under the project folder.

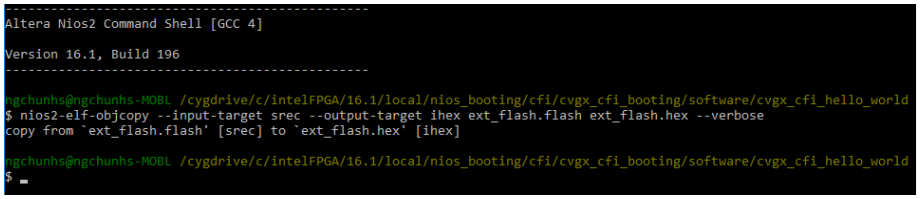
Figure 177. Selecting mem_init_generate in Make Targets



5. Launch **Nios II Command Shell**. Use **nios2-elf-objcopy** utility to convert the **.flash** file to **HEX** file.

```
nios2-elf-objcopy --input-target srec --output-target ihex <*.flash>
<*.hex> --verbose
```

Figure 178. FLASH to HEX Conversion



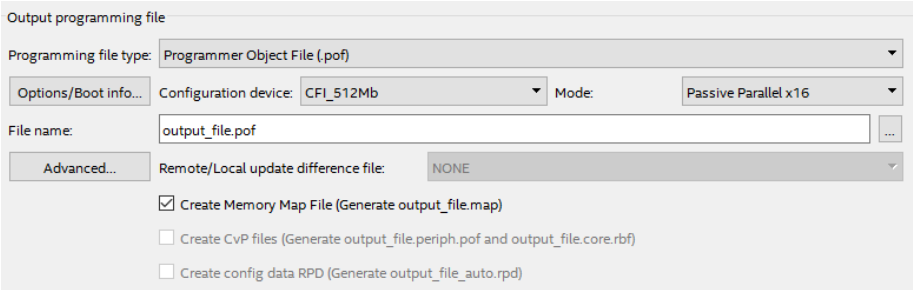
5.2.6.2.3. Programming

Programming Files Generation

1. Open the **Convert Programming Files** from **File** menu in Intel Quartus Prime software.
2. Under **Output programing file** section, set the following items:

a. **Programming file type**: Programmer Object File (.pof)
b. **Configuration device**: Select according to your CFI flash
c. **Mode**: Passive Parallel x8/x16/32[20]
d. **File name**: You may select your preferred path for the output file (.pof). By default this is generated under your project root directory.
e. Keep the default settings for **Create Memory Map File**.

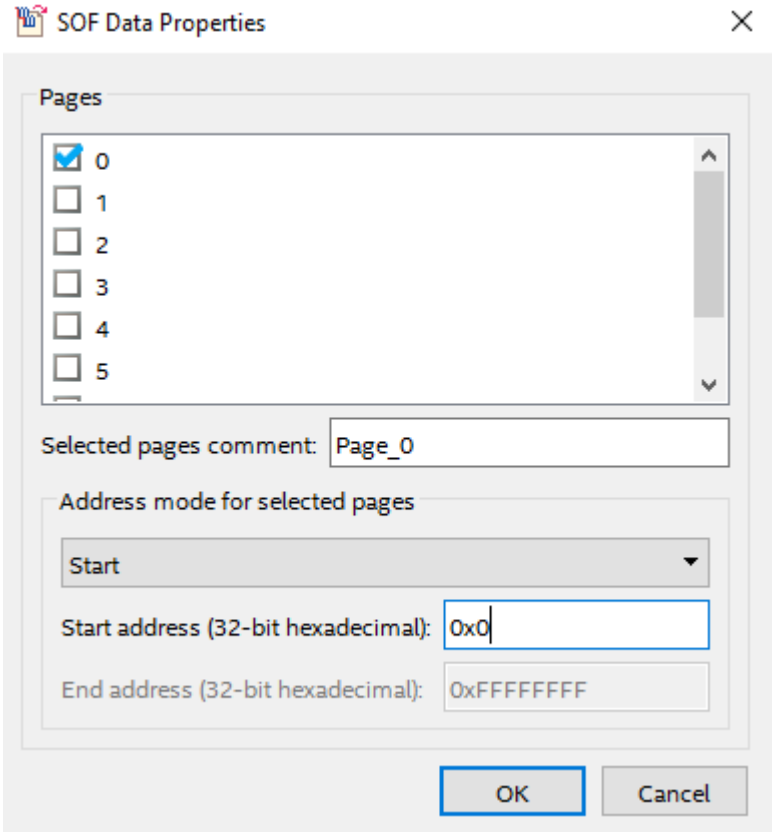
Figure 179. Output programming file Setting



3. Under **Input files to convert** section:
   a. Select **SOF Data**, then **Add File...** and choose the ***.sof** file generated by Intel Quartus Prime compilation.
   b. Select the added ***.sof** file, click **Properties**, then enable **Compression** and click **OK**.
   c. Select **SOF Data** and click **Properties**. Set the **Address mode for selected pages** to **Start** and set the **Start address (32-bit hexadecimal)**, then click **OK**.

(20) Set data width according to the supported scheme.

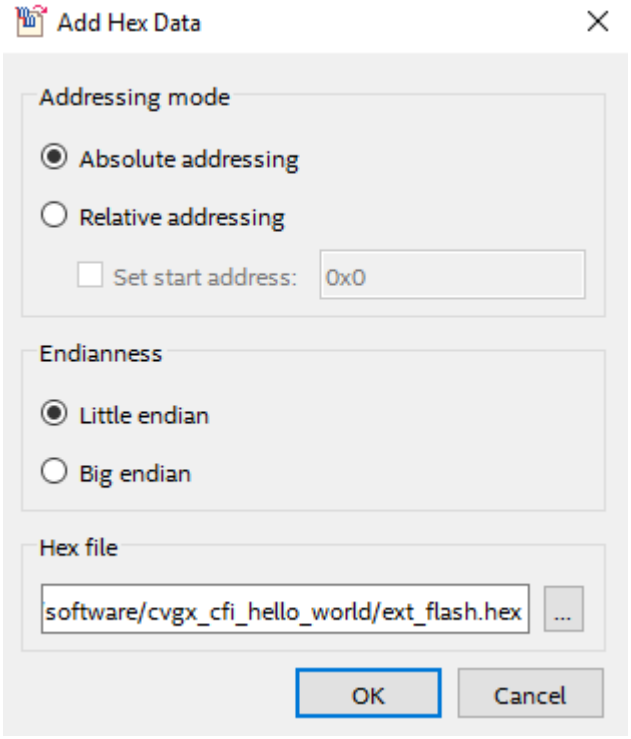
Figure 180. SOF Data Properties



d. Click **Add Hex Data** and select **Absolute addressing** mode. Browse to the HEX file location and click **OK**.

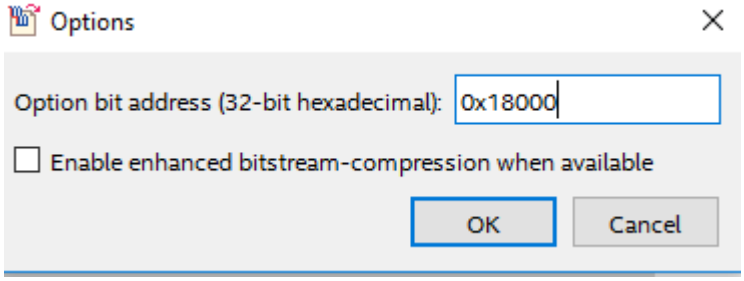
Note: You may select Relative addressing mode if you would like to set a relative address to the reset vector offset you configured earlier.

Figure 181. Add Hex Data



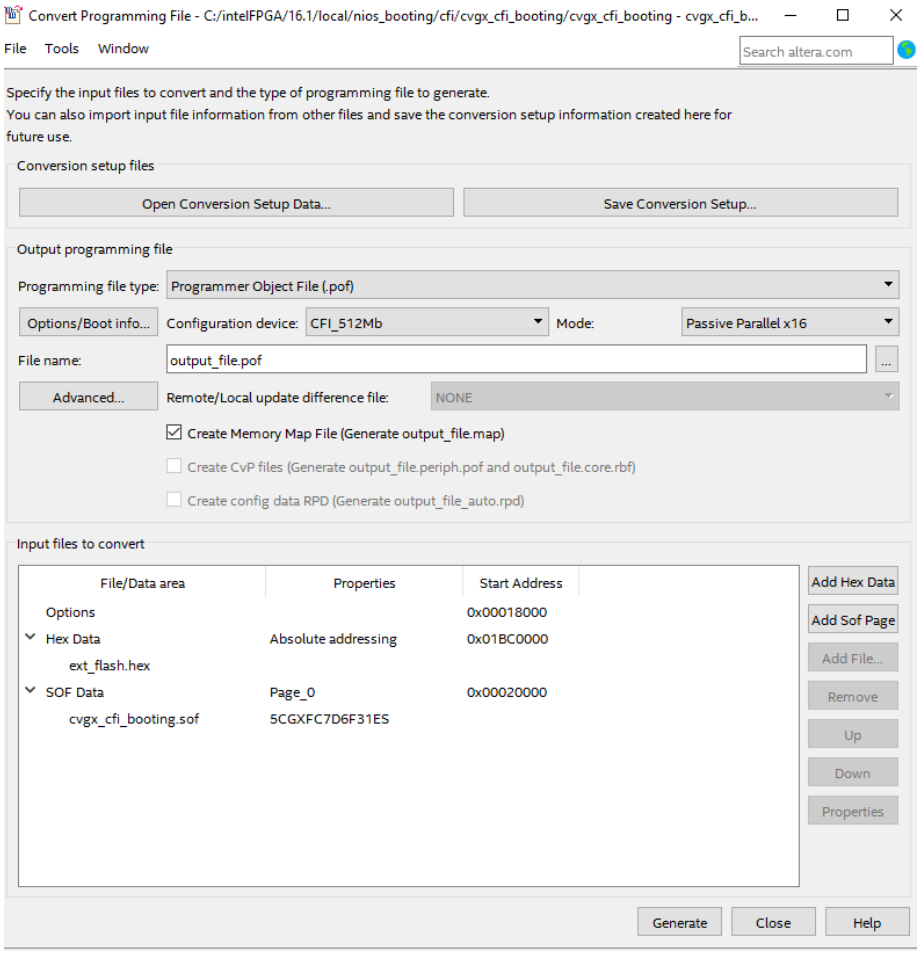
4. Click **Options/Boot info.**, set the **Option bit address (32-bit hexadecimal)**.

Figure 182. Option Bit Address



5. Click **Generate** to generate the POF programming file.

Figure 183. Convert Programming File Settings

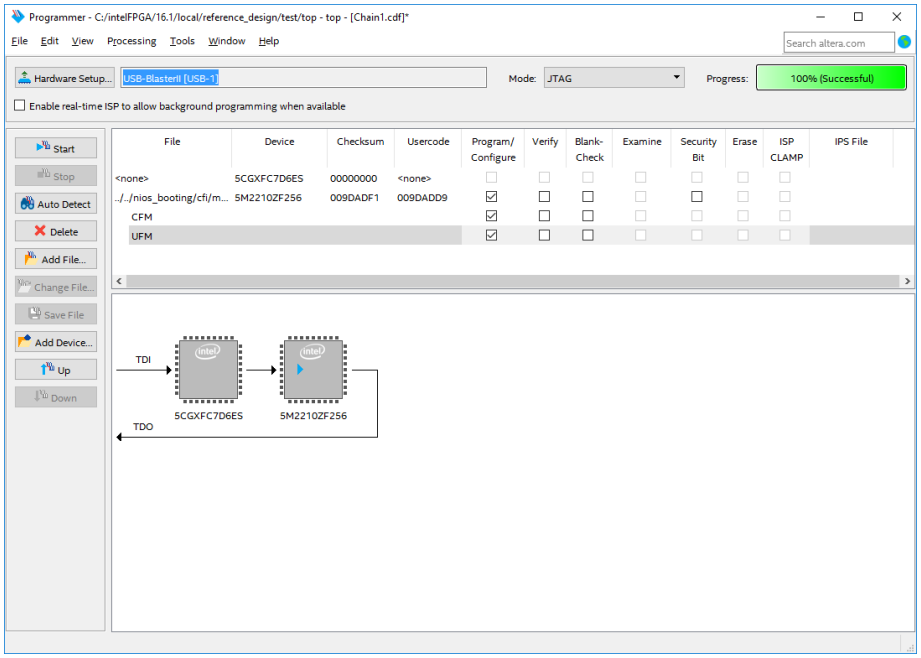


CFI Flash Programming

1. Ensure the MSEL pin setting on your board is configured to **Fast Passive Parallel (FPP) x8/x16/32**[21].
2. Open the Intel Quartus Prime programmer and make sure JTAG was detected under the **Hardware Setup**.
3. Click **Auto Detect** and select your FPGA device, then click **OK**.
4. Right click on the MAX CPLD and click **Change File**.
5. Open the PFL ***.pof** file that contains programming logic.
6. Check the MAX CPLD including **CFM** and **UFM** under **Program/Configure**.
7. Click **Start** to start programming the MAX CPLD.

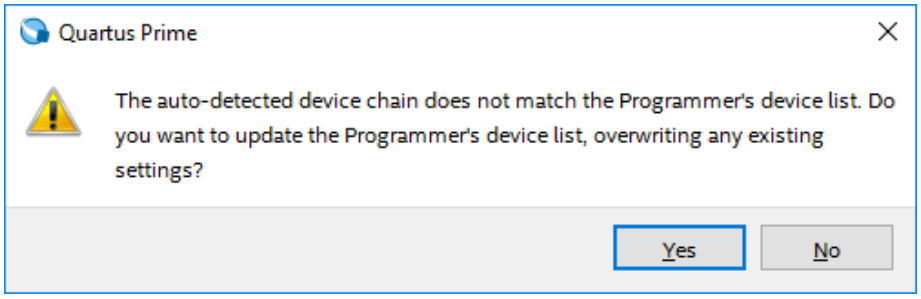
(21) Set data width according to supported scheme.

Figure 184. Program MAX CPLD with Programming Logic



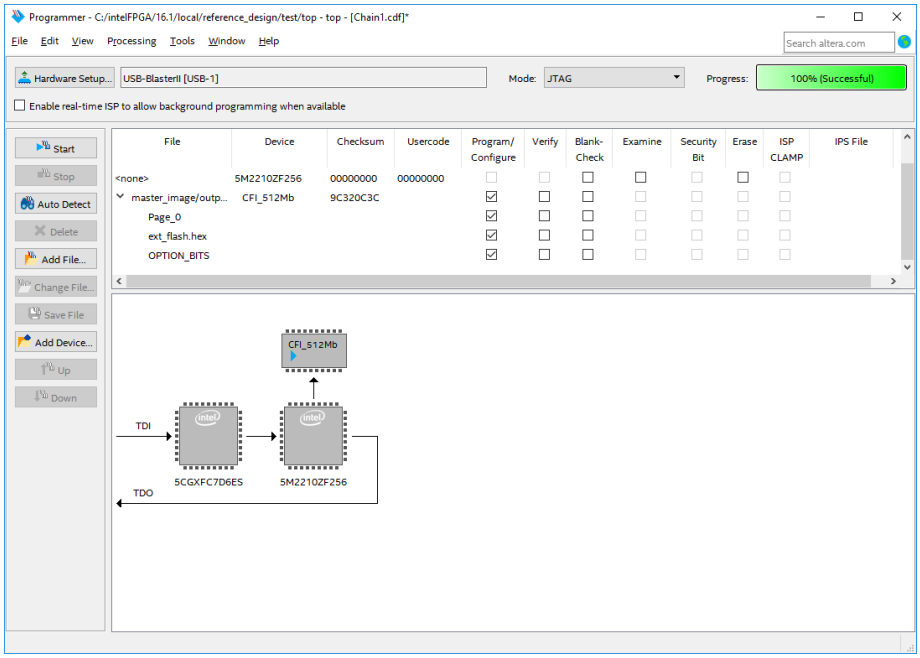
8. Click **Auto Detect** after the programming successful. Click **Yes** if you are asked to overwrite existing settings.

Figure 185. Overwrite Existing Settings



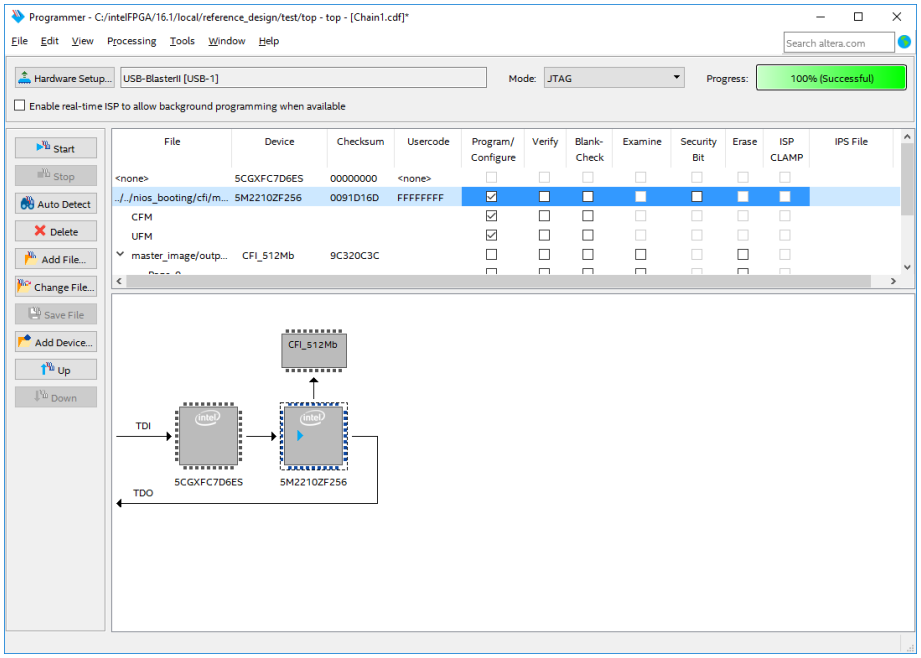
9. The CFI flash device will be attached to the MAX CPLD. Right click on the CFI flash device and click **Change File**.
10. Open the generated ***.pof** file that contains Nios II application.
11. Check the CFI flash device including **Page_0**, **HEX file** and **OPTION_BITS** under **Program/Configure**.\
12. Click **Start** to start programming the CFI flash.

Figure 186. Program CFI Flash



13. After the programming successful, right click on the MAX CPLD and click **Change File**.
14. Open the PFL ***.pof** file that contains configuration controller logic.
15. Check the MAX CPLD including **CFM** and **UFM** under **Program/Configure**.
16. Click **Start** to start programming the MAX CPLD.

Figure 187. Program MAX CPLD with Configuration Controller Logic



17. Power cycle your hardware.

5.2.6.3. Nios II Processor Application Copied from CFI Flash to RAM using Boot Copier

Using a boot copier to copy the Nios II application from CFI flash to RAM is suitable when multiple iterations of application software development and high system performance are required. The Nios II SBT tool automatically adds the Nios II processor memcpy-based boot copier to the system when the executable file (.elf) is converted to the memory initialization file (.hex). The boot copier is located at the base address of the HEX data, followed by the application.

For this boot option, the Nios II processor starts executing the boot copier software upon system reset that copies the application from the CFI to the on-chip memory or External RAM. Once this process is complete, the Nios II processor transfers the program control over to the application.

Figure 188. Nios II Application Copied from CFI Flash to RAM using Boot Copier

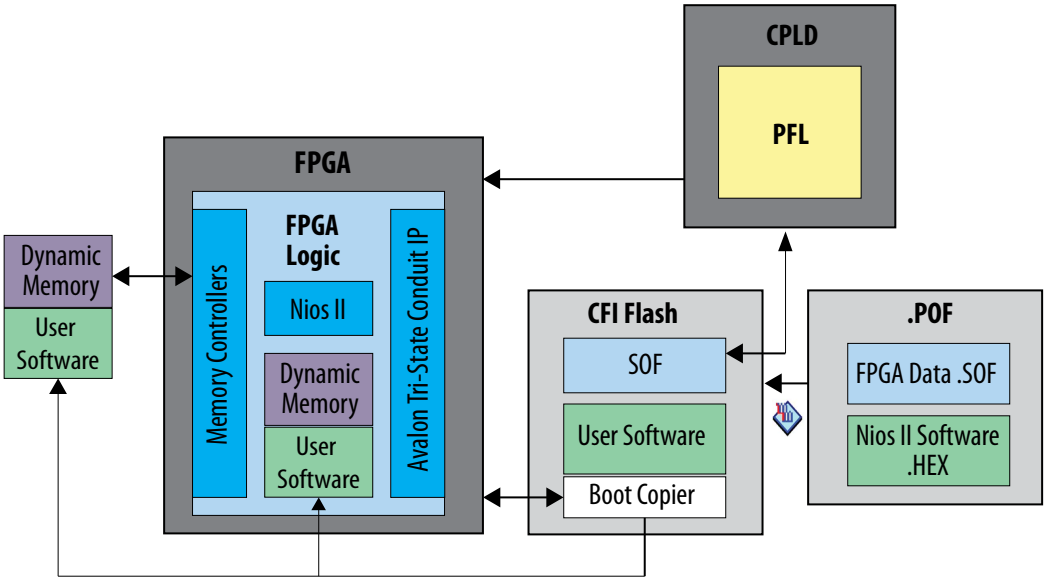
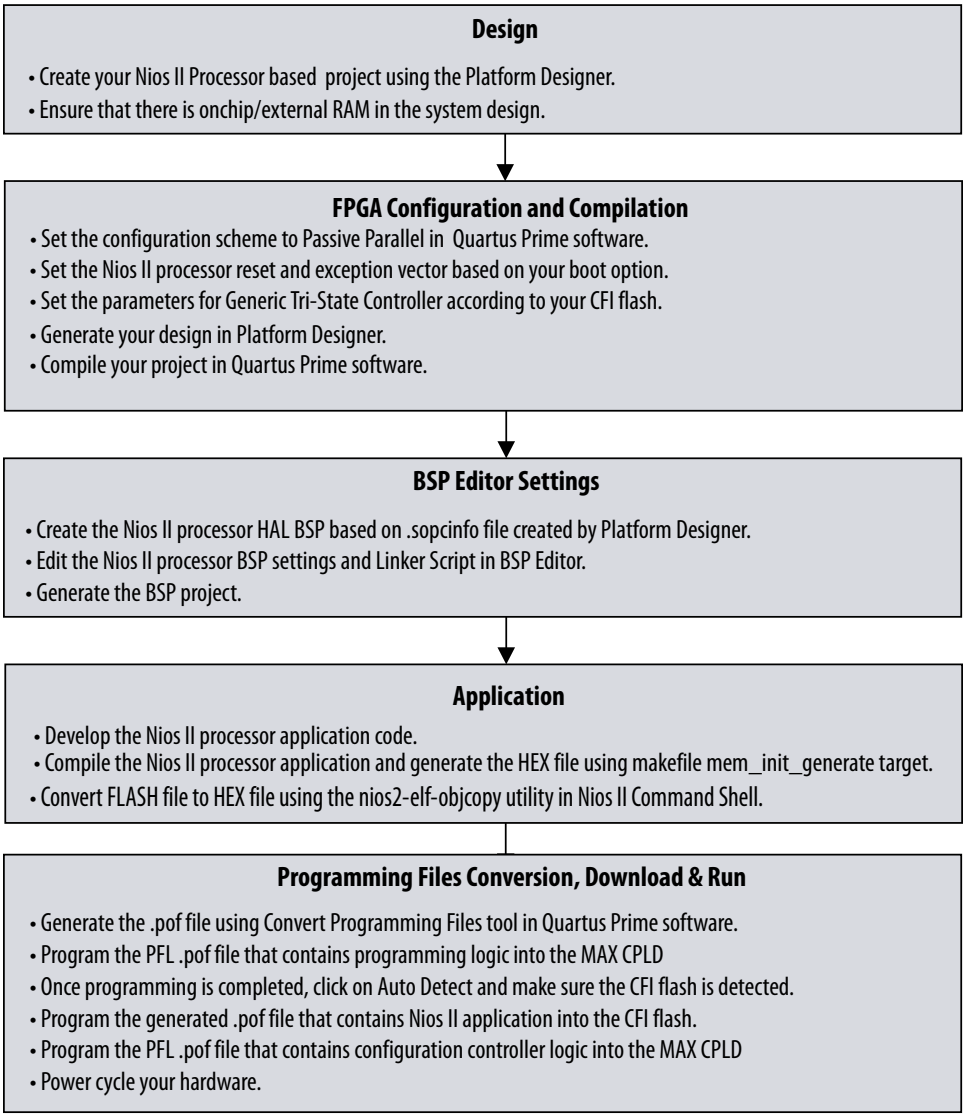


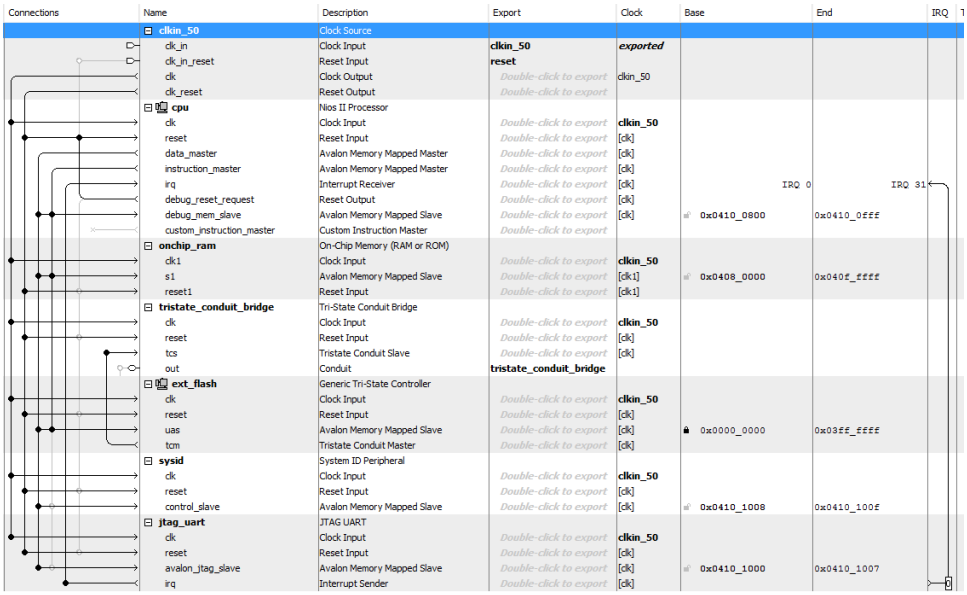
Figure 189. Design, Configuration and Booting Flow



5.2.6.3.1. Hardware Design

1. Create your Intel Quartus Prime project and Platform Designer design.
2. Make sure OCRAM or External RAM is added into your Platform Designer system.
3. Refer to diagram below for example IP connections in Platform Designer.

Figure 190. Example IP Connections in Platform Designer for Booting Nios II from CFI Flash



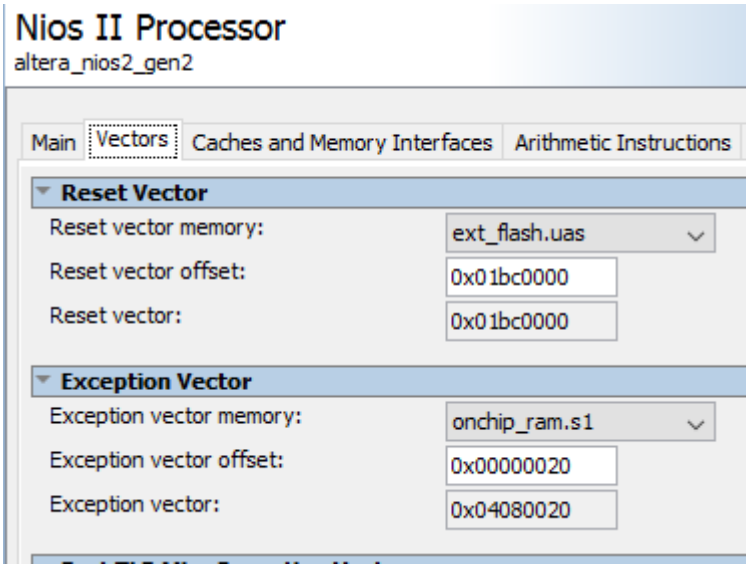
IP Component Settings

1. In the Nios II processor parameter editor, set the **Reset vector memory** to CFI flash and **Exception vector memory** to OCRAM / External RAM.
   a. The **Reset vector** must be the base address of your application. In this example, it is 0x01bc0000.

      Note: Your ***.sof** image size influences your reset vector offset configuration. The reset vector offset is the start address of the **.hex** file in CFI flash and it must point to a location after the ***.sof** image. You can determine the minimum reset vector offset by using the following equation:

```
minimum reset vector offset = (.sof image start address + .sof image
size) in HEX
```

Figure 191. Nios II Parameter Editor Settings with Exception Vector Set to OCRAM / External RAM



2. In Generic Tri-State Controller IP parameter editor, select one of the preset configurations in **Library** to automatically assign the parameter values.

Figure 192. Generic Tri-State Controller Preset Configuration



If your flash device is not listed in the **Library**, you can use the parameter editor to specify the required settings.

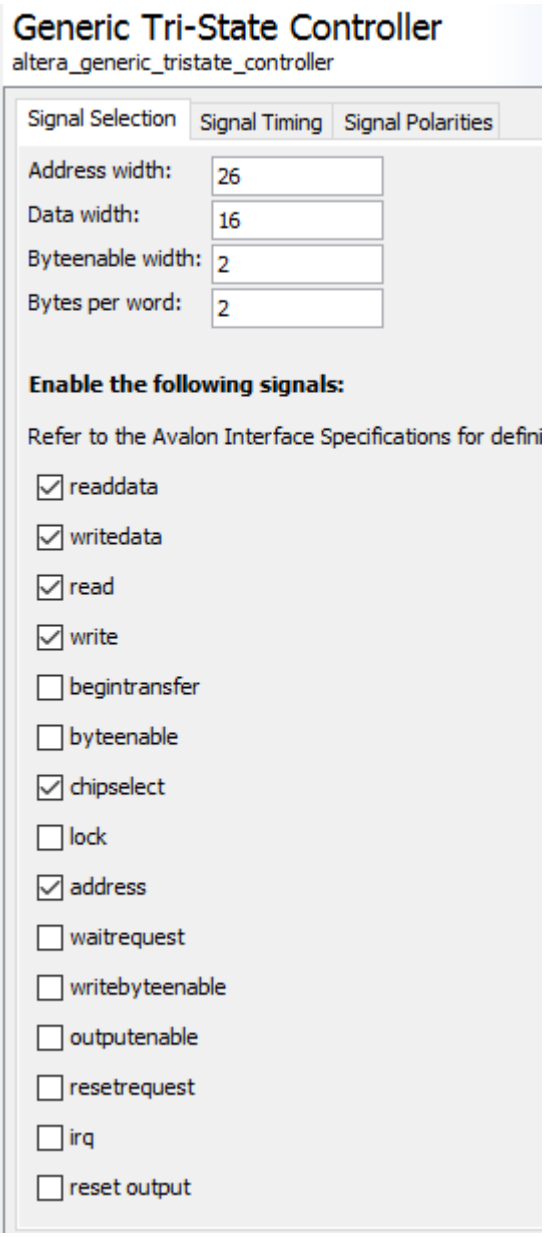
Note: The appropriate values for these parameters are typically listed in the device's datasheet.

On top of the device specific parameters, ensure the following settings are correct in place:

a. In **Signal Selection** tab, enable the following signals:

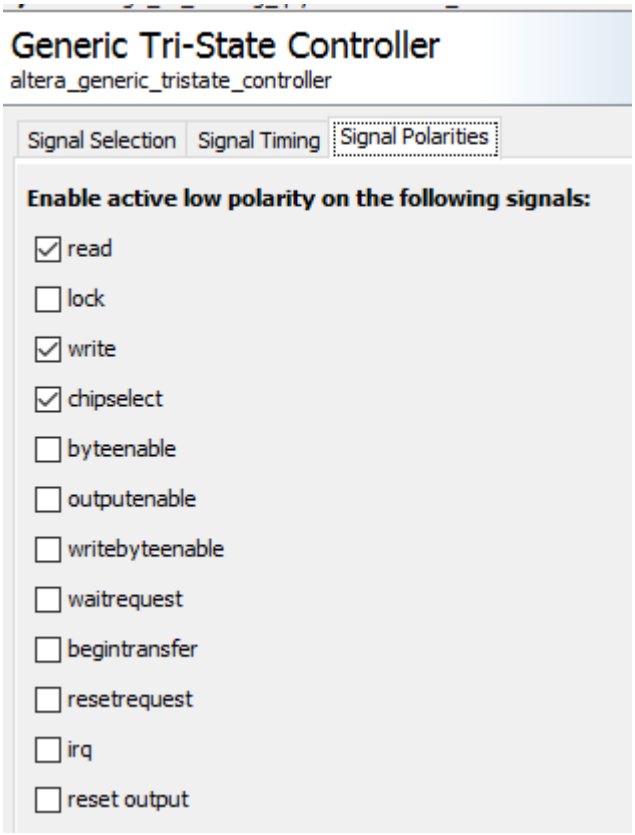
- `readdata`
- `writedata`
- `read`
- `write`
- `chipselect`
- `address`

Figure 193. Signal Selection



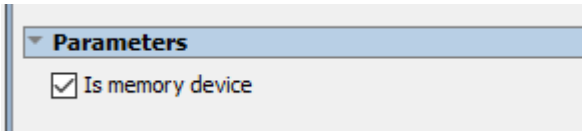
b. In **Signal Polarities** tab, enable the following signals:
   - `read`
   - `write`
   - `chipselect`

Figure 194. Signal Polarities



c. In **Parameters** tab, select **Is memory device**.

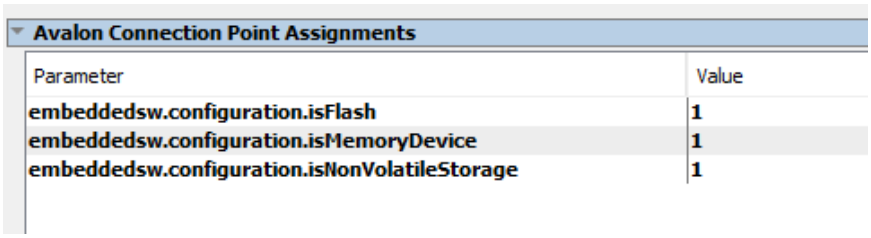
Figure 195. Parameters Window



d. In **Avalon Connection Point Assignments** tab, set the following parameters to a value of 1:

- **embeddedsw.configuration.isFlash**
- **embeddedsw.configuration.isMemoryDevice**
- **embeddedsw.configuration.isNonVolatileStorage**

Figure 196. Avalon Connection Point Assignments

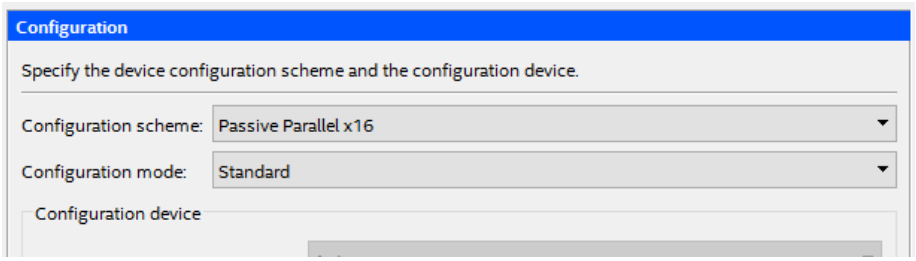


3. Click **Generate HDL**. The **Generation** dialog box appears.
4. Specify output file generation options, and then click **Generate**.

Intel Quartus Prime Software Settings

1. In the Intel Quartus Prime software, click on **Assignments > Device > Device and Pin Options > Configuration**. Set **Configuration scheme** to **Passive Parallel x8/x16/x32**(22)

Figure 197. Configuration Scheme Selection



2. Click **OK** to exit the **Device and Pins Options** window.
3. Click **OK** to exit the **Device** window.
4. Click on **Processing > Start Compilation** to compile your project and generate the .sof file.

5.2.6.3.2. Application

In the Nios II SBT window, select:

- **File ➤ New ➤ Nios II Application** to develop a Nios II application.
- **File ➤ New ➤ Nios II Application and BSP from Template** to create a template to use for the Nios II application.

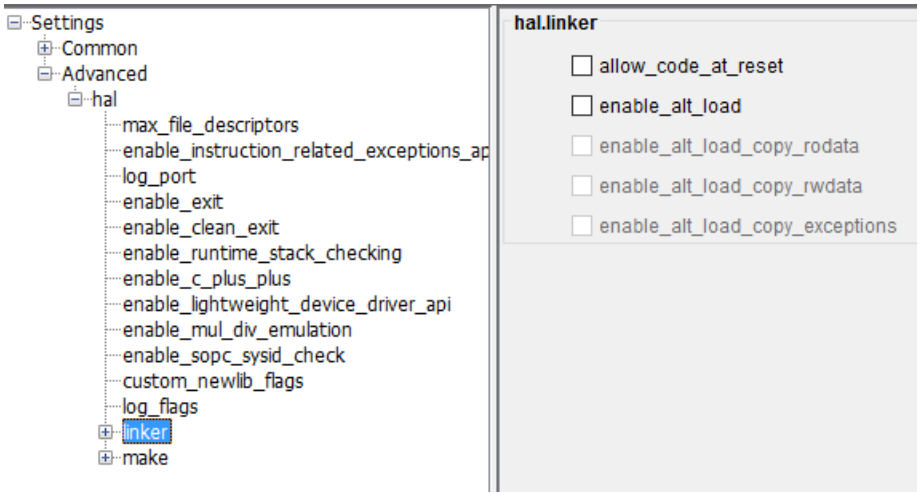
BSP Editor Settings

You must edit the BSP settings according to the selected Nios II processor boot options.

(22) Set data width according to supported scheme.

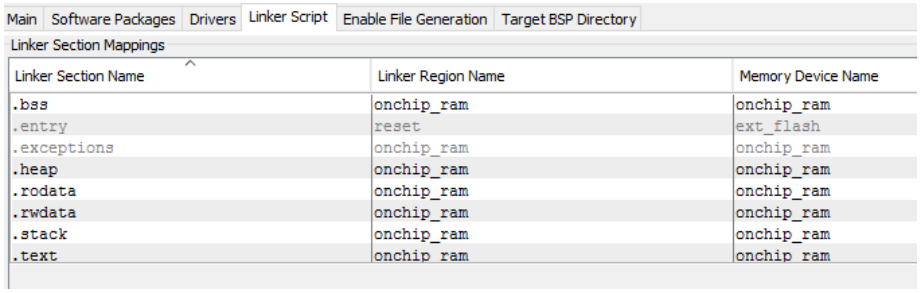
1. In the Nios II SBT tool, right click on your BSP project in the **Project Explorer** window. Select **Nios II > BSP Editor...** to open the Nios II BSP Editor.
2. In Nios II BSP Editor, expand into **Advanced.hal.linker** under **Settings** in **Main** tab.
3. Leave all settings unchecked.

Figure 198. Advanced.hal.linker Settings



4. Click on the **Linker Script** tab in the Nios II BSP Editor.
5. Set all regions in the **Linker Section Name** list to the OCRAM or External RAM.

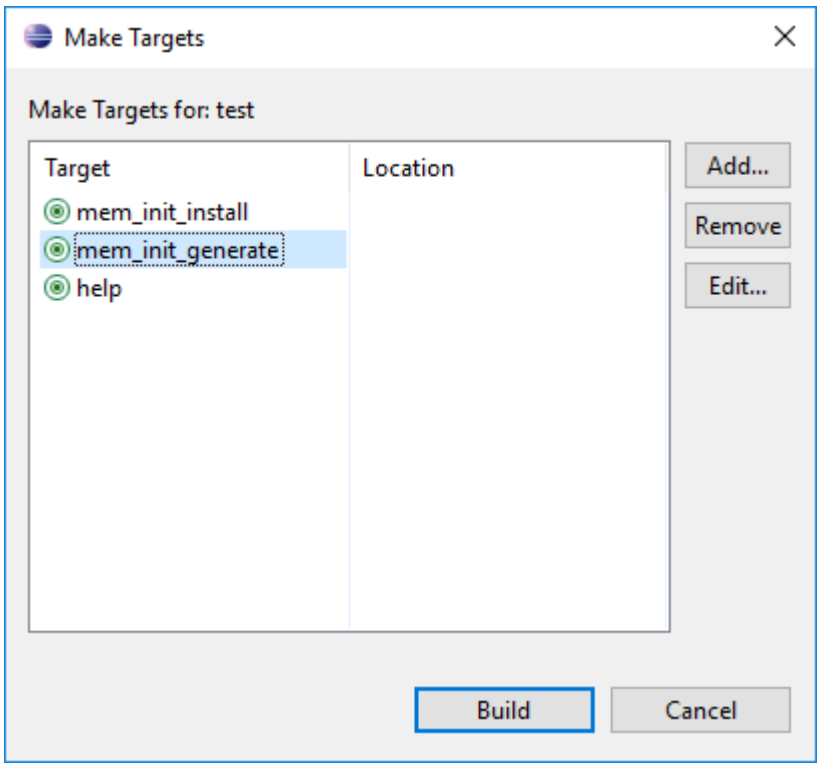
Figure 199. Linker Region Settings



HEX File Generation

1. After creating the Nios II application, right click on the project in **Project Explorer** and select **Build Project**. An ***.elf** file is created under the project folder.
2. Right click again on your project and click on **Make Targets** > **Build...**, the **Make Targets** dialog box appears. You can also press **shift + F9** to trigger the **Make Targets** dialog box.
3. Select **mem_init_generate**.
4. Click **Build**. A ***.flash** file is created under the project folder.

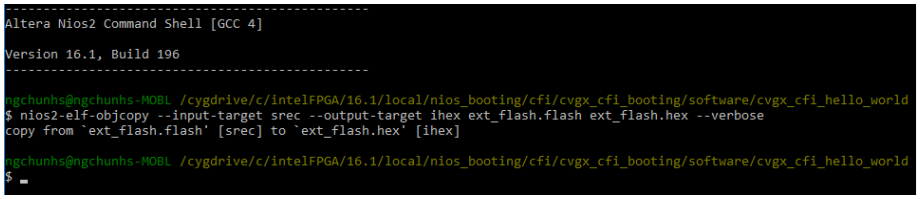
Figure 200. Selecting mem_init_generate in Make Targets



5. Launch **Nios II Command Shell**. Use **nios2-elf-objcopy** utility to convert the **.flash** file to **HEX** file.

```
nios2-elf-objcopy --input-target srec --output-target ihex <*.flash>
<*.hex> --verbose
```

Figure 201. FLASH to HEX Conversion



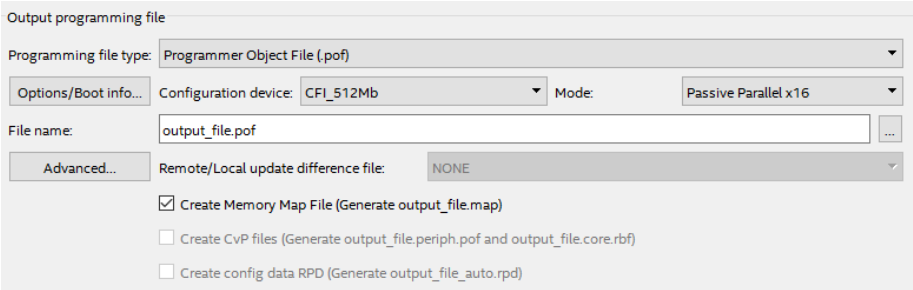
5.2.6.3.3. Programming

Programming Files Generation

1. Open the **Convert Programming Files...** from **File** menu in Intel Quartus Prime software.
2. Under **Output programing file** section, set the following items:

a. **Programming file type**: Programmer Object File (.pof)
b. **Configuration device**: Select according to your CFI flash
c. **Mode**: Passive Parallel x8/x16/32[23]
d. **File name**: You may select your preferred path for the output file (.pof). By default this is generated under your project root directory.
e. Keep the default settings for **Create Memory Map File**.

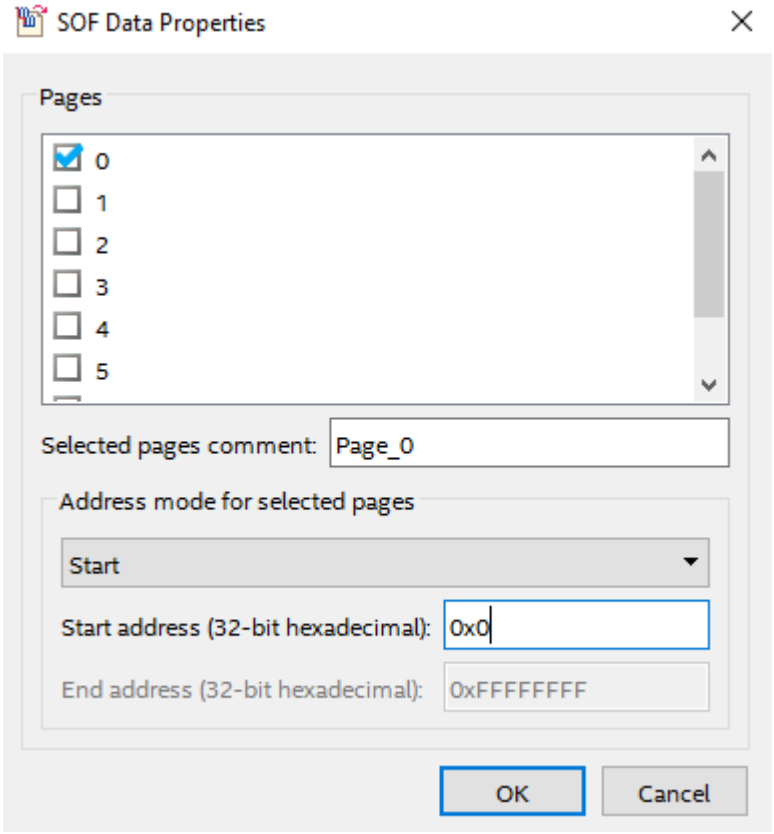
Figure 202. Output programming file Setting



3. Under **Input files to convert** section:
   a. Select **SOF Data**, then **Add File...** and choose the ***.sof** file generated by Intel Quartus Prime compilation.
   b. Select the added ***.sof** file, click **Properties**, then enable **Compression** and click **OK**.
   c. Select **SOF Data** and click **Properties**. Set the **Address mode for selected pages** to **Start** and set the **Start address (32-bit hexadecimal)**, then click **OK**.

(23) Set data width according to the supported scheme.

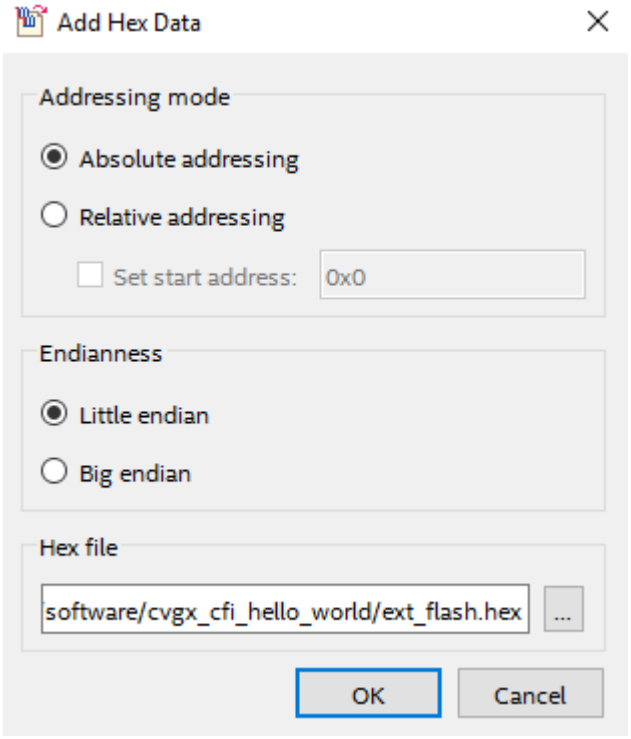
Figure 203. SOF Data Properties



d. Click **Add Hex Data** and select **Absolute addressing** mode. Browse to the HEX file location and click **OK**.

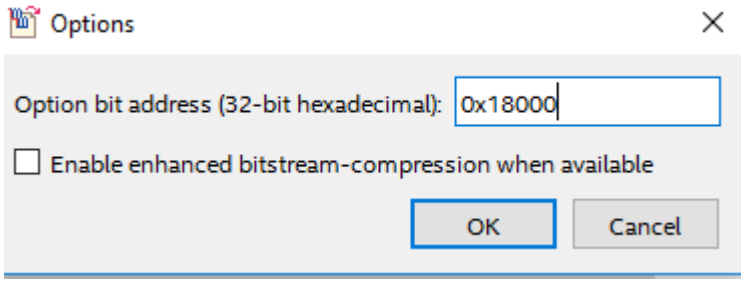
Note: You may select Relative addressing mode if you would like to set a relative address to the reset vector offset you configured earlier.

Figure 204. Add Hex Data



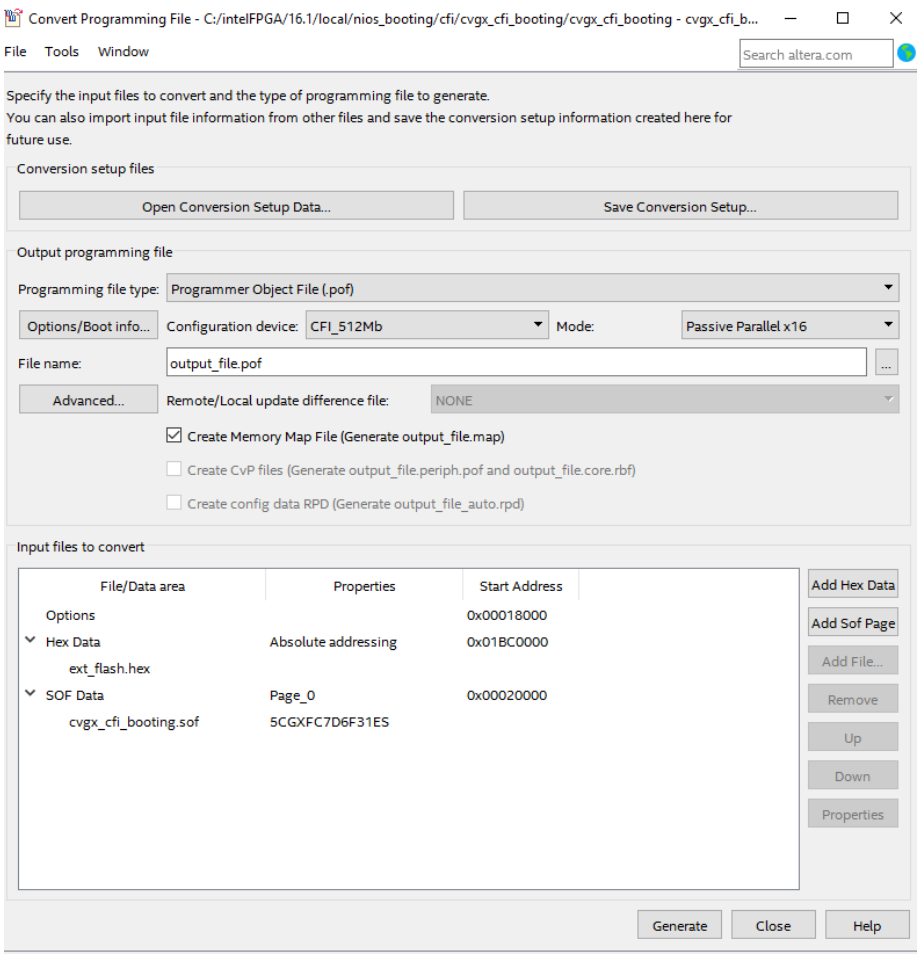
4. Click **Options/Boot info.**, set the **Option bit address (32-bit hexadecimal)**.

Figure 205. Option Bit Address



5. Click **Generate** to generate the POF programming file.

Figure 206. Convert Programming File Settings

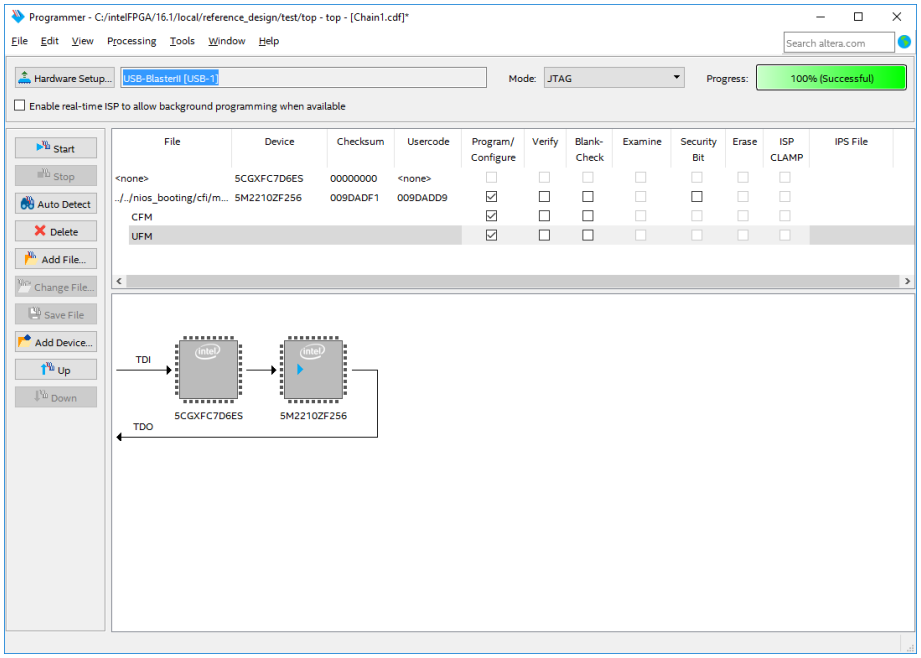


CFI Flash Programming

1. Ensure the MSEL pin setting on your board is configured to **Fast Passive Parallel (FPP) x8/x16/32**[24].
2. Open the Intel Quartus Prime programmer and make sure JTAG was detected under the **Hardware Setup**.
3. Click **Auto Detect** and select your FPGA device, then click **OK**.
4. Right click on the MAX CPLD and click **Change File**.
5. Open the PFL ***.pof** file that contains programming logic.
6. Check the MAX CPLD including **CFM** and **UFM** under **Program/Configure**.
7. Click **Start** to start programming the MAX CPLD.

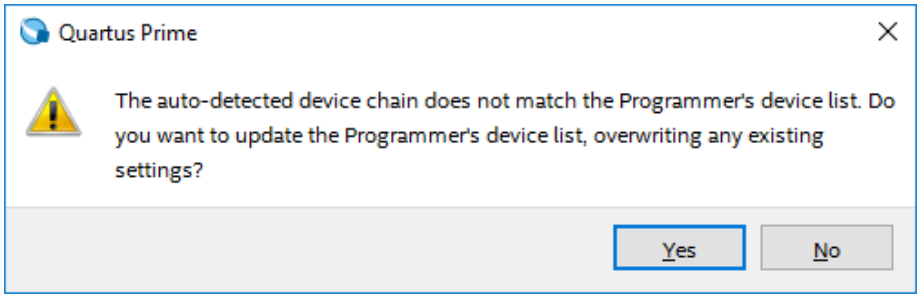
(24) Set data width according to supported scheme.

Figure 207. Program MAX CPLD with Programming Logic



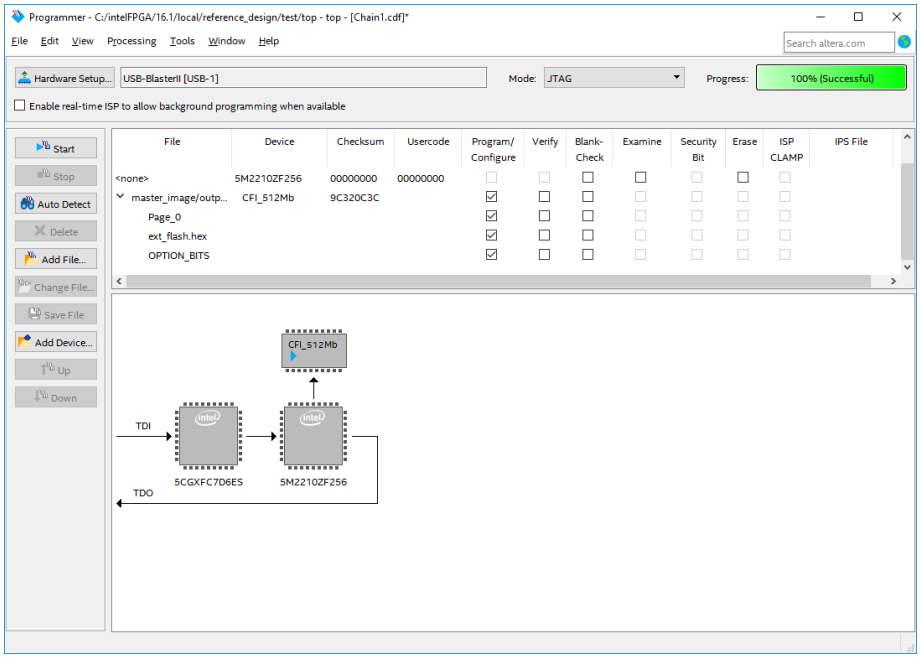
8. Click **Auto Detect** after the programming successful. Click **Yes** if you are asked to overwrite existing settings.

Figure 208. Overwrite Existing Settings



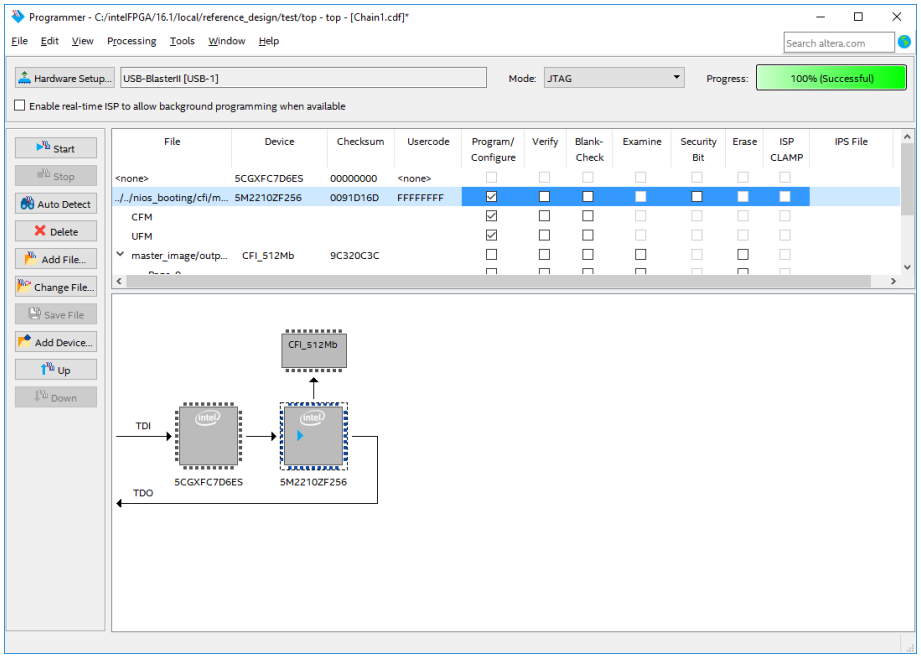
9. The CFI flash device will be attached to the MAX CPLD. Right click on the CFI flash device and click **Change File**.
10. Open the generated ***.pof** file that contains Nios II application.
11. Check the CFI flash device including **Page_0**, **HEX file** and **OPTION_BITS** under **Program/Configure**.\
12. Click **Start** to start programming the CFI flash.

Figure 209. Program CFI Flash



13. After the programming successful, right click on the MAX CPLD and click **Change File**.
14. Open the PFL ***.pof** file that contains configuration controller logic.
15. Check the MAX CPLD including **CFM** and **UFM** under **Program/Configure**.
16. Click **Start** to start programming the MAX CPLD.

Figure 210. Program MAX CPLD with Configuration Controller Logic



17. Power cycle your hardware.

5.2.7. Summary of Nios II Processor Vector Configurations and BSP Settings

The following table shows a summary of Nios II processor reset and exception vector configurations, and BSP settings.

Table 40. Summary of Nios II Processor Vector Configurations and BSP Settings

Boot Option	Reset Vector Configuration	Exception Vector Configuration	BSP Editor Setting: Settings.Advanced.hal.linker	BSP Editor Setting: Linker Script
Nios II processor application execute-in-place from On-chip Flash (UFM)	On-chip Flash (UFM)	Choose between: • OCRAM/ External RAM • On-chip Flash	If the exception vector memory is set to OCRAM/ External RAM, enable the following settings in **Settings.Advanced.hal.linker**: • `allow_code_at_reset` • `enable_alt_load` • `enable_alt_load_copy_rodata` • `enable_alt_load_copy_rwdata` • `enable_alt_load_copy_exceptions` If the exception vector memory is set to On-chip Flash, enable the following settings in **Settings.Advanced.hal.linker**: • `allow_code_at_reset` • `enable_alt_load` • `enable_alt_load_copy_rodata` • `enable_alt_load_copy_rwdata`	• Set **.text** Linker Section to UFM • Set other Linker Sections (**.heap, .rwdata, .rodata, .bss, .stack**) to OCRAM / External RAM
Nios II processor application copied from UFM to RAM using boot copier	On-chip Flash (UFM)	OCRAM/ External RAM	Make sure all settings in **Settings.Advanced.hal.linker** are left unchecked.	Make sure all Linker Sections are set to OCRAM / External RAM.
Nios II processor application execute-in-place from EPCQ flash	EPCQ Flash	Choose between: • OCRAM/ External RAM • EPCQ Flash	If the exception vector memory is set to OCRAM/ External RAM, enable the following settings in **Settings.Advanced.hal.linker**: • `allow_code_at_reset` • `enable_alt_load` • `enable_alt_load_copy_rodata` • `enable_alt_load_copy_rwdata` • `enable_alt_load_copy_exceptions`	• Set **.text** Linker Section to EPCQ flash • Set other Linker Sections (**.heap, .rwdata, .rodata, .bss, .stack**) to OCRAM / External RAM

continued...

Boot Option	Reset Vector Configuration	Exception Vector Configuration	BSP Editor Setting: Settings.Advanced.hal.linker	BSP Editor Setting: Linker Script
			If the exception vector memory is set to EPCQ Flash, enable the following settings in **Settings.Advanced.hal.linker**: • `allow_code_at_reset` • `enable_alt_load` • `enable_alt_load_copy_rodata` • `enable_alt_load_copy_rwdata`	
Nios II processor application copied from EPCQ flash to RAM using boot copier	EPCQ Flash	OCRAM/ External RAM	Make sure all settings in **Settings.Advanced.hal.linker** are left unchecked.	Make sure all Linker Sections are set to OCRAM / External RAM.
Nios II processor application executes in-place from QSPI flash	QSPI Flash	Choose between: • OCRAM/ External RAM • QSPI Flash	If the exception vector memory is set to OCRAM/ External RAM, enable the following settings in **Settings.Advanced.hal.linker**: • `allow_code_at_reset` • `enable_alt_load` • `enable_alt_load_copy_rodata` • `enable_alt_load_copy_rwdata` • `enable_alt_load_copy_exceptions` If the exception vector memory is set to QSPI Flash, enable the following settings in **Settings.Advanced.hal.linker**: • `allow_code_at_reset` • `enable_alt_load` • `enable_alt_load_copy_rodata` • `enable_alt_load_copy_rwdata`	• Set **.text** Linker Section to QSPI flash • Set other Linker Sections (**.heap, .rwdata, .rodata, .bss, .stack**) to OCRAM / External RAM
Nios II processor application copied from QSPI flash to RAM using boot copier	QSPI flash	OCRAM/External RAM	Make sure all settings in **Settings.Advanced.hal.linker** are left unchecked	Make sure all Linker Sections are set to OCRAM / External RAM.
Nios II processor application execute-in-place from On-chip Memory (OCRAM)	OCRAM	OCRAM	Enable **allow_code_at_reset** in **Settings.Advanced.hal.linker** and other settings are left unchecked.	Make sure all Linker Sections are set to OCRAM.
Nios II processor application execute-in-place from CFI flash	CFI Flash	Choose between: • OCRAM/ External RAM • CFI Flash	If the exception vector memory is set to OCRAM/ External RAM, enable the following settings in **Settings.Advanced.hal.linker**:	• Set **.text** Linker Section to CFI flash • Set other Linker Sections (**.heap, .rwdata, .rodata, .bss, .stack**) to OCRAM / External RAM

continued...

Boot Option	Reset Vector Configuration	Exception Vector Configuration	BSP Editor Setting: Settings.Advanced.hal.linker	BSP Editor Setting: Linker Script
			• `allow_code_at_reset` • `enable_alt_load` • `enable_alt_load_copy_rodata` • `enable_alt_load_copy_rwdata` • `enable_alt_load_copy_exceptions` If the exception vector memory is set to CFI Flash, enable the following settings in **Settings.Advanced.hal.linker**: • `allow_code_at_reset` • `enable_alt_load` • `enable_alt_load_copy_rodata` • `enable_alt_load_copy_rwdata`	
Nios II processor application copied from CFI flash to RAM using boot copier	CFI Flash	OCRAM/External RAM	Make sure all settings in **Settings.Advanced.hal.linker** are left unchecked	Make sure all Linker Sections are set to OCRAM / External RAM.

5.3. Alternative Nios II Boot Methods

In any stand-alone embedded system that contains a microprocessor, the processor runs a small piece of code called a boot copier, or boot loader, after the system resets. The boot copier locates the appropriate application software in non-volatile memory, copies it to RAM, initializes critical system components, and branches to the entry point of the application software image. The block of data in non-volatile memory that contains the application software is commonly referred to as the boot image. Boot copiers range in complexity from basic byte-for-byte copy routines to comprehensive applications that perform rigorous system tests, select among multiple software applications, and unpack, decompress, and perform error detection on the proper application.

This document teaches you how to implement your own custom boot copier software using the Nios II processor and Nios II software build tools. In addition, this document provides the basic information needed to externally control the Nios II boot process.

This document addresses how to implement a custom boot copier for a Nios II processor already configured in the FPGA. It does not address custom methods of configuring Intel FPGAs.

For information about custom methods for configuring Intel FPGAs, refer to the "Configuration Solutions" web page on the Intel FPGA website.

5.3.1. Assumptions About the Reader

This document assumes that you are an advanced Nios II user and that you are comfortable reading and writing embedded software. If you are not familiar with the Nios II hardware or software development flow, Intel strongly recommends that you first become familiar with building a Nios II microprocessor system.

For step-by-step procedures that build an example Nios II microprocessor system, refer to the *Nios II Gen2 Hardware Development Tutorial*.

This document also assumes you are familiar with the command line operation of the Nios II flash programmer.

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

5.3.2. Implementing a Custom Boot Copier

Implementing a custom boot copier requires you to deviate from the normal Nios II software build tools development flow. You must edit source files by hand and run file conversion utilities from the Nios II command shell.

This document includes example source code for the following types of custom boot copiers:

- Advanced boot copier—Includes extra features such as dual boot image support and CRC error checking.
- Small boot copier—Is a bare-minimum boot copier that requires very little memory space.

The files referenced in this application note are available in two **.zip** files on the Intel website. One file contains the hardware design example and the other contains the software program files.

5.3.2.1. Hardware Design Files

The "Nios II Ethernet Standard Design Example" page of the Intel FPGA website contains the hardware design files for a number of Intel development boards. Navigate to the web page and locate the Nios II Ethernet Standard design example **.zip** file that corresponds to your board. Download and unzip the file in a project directory of your choosing. The remainder of this application note refers to this directory as <*project*>.

For more information about the design example and supported Intel development boards, refer to the *Nios II Ethernet Standard Design Example* located on the "Design Examples" web page on the Intel FPGA website.

5.3.2.2. Software Files

The design files for this section contains the software program files and board support package (BSP). Download and unzip the file into your <*project*> directory.

Related Information

Download Design Files

5.3.3. Default Nios II Boot Copier

This section discusses the operation of the default Nios II boot copier, describing the workings of both the Common Flash Interface (CFI) flash memory and the Intel erasable programmable configurable serial (EPCS) or quad serial configuration (EPCQ) variant. If you are unfamiliar with the default boot copier, read this section before implementing a custom boot copier.

5.3.3.1. Overview of the Default Nios II Boot Copier

The default boot copier included with the Nios II processor provides sufficient functionality for most Nios II applications and is convenient to implement with the Nios II software build tools development flow. The default boot copier is automatically and transparently added to your system when you convert your executable files to flash programming files.

Note: Intel recommends that you use the default Nios II boot copier unless you require a custom boot copier with different or additional functionality. Implementing a custom boot copier can complicate your software build process and hinder Intel's ability to provide technical support.

The default Nios II boot copier has the following features:

- Supports CFI or EPCS/EPCQ flash memory
- Unpacks and copies boot image to RAM
- Automatically branches to application code in RAM

5.3.3.2. The Default CFI Flash Boot Copier

The Nios II default boot copier is automatically included by the Nios II flash programmer during the execution of the **elf2flash** utility. Based on the processor reset address, the **elf2flash** utility determines the entry point of the application code and the address range of flash memory, whether or not a boot copier is needed. A CFI boot copier is needed whenever the processor's reset address points to CFI flash memory and the application's .text section points to somewhere other than CFI flash memory. When a boot copier is needed, **elf2flash** packs the application code in a boot record, and then creates a Motorola S-record Flash Programming File (**.flash**) containing the default boot copier and the boot record. The flash programmer downloads this boot record to CFI flash memory.

Immediately after the Nios II processor completes reset, the boot copier executes, reads the boot record as described in the "Boot Images" section, and copies the application code to RAM. After copying is complete, the boot copier reads the entry point of the application code from the boot record. The boot copier executes the jump to that address, and the application software begins executing.

5.3.3.3. The Default EPCS/EPCQ Boot Copier

When the Nios II processor reset address is set to the base address of an EPCS/EPCQ controller in the Platform Designer system integration tool, the default EPCS/EPCQ boot copier is implemented. The EPCS/EPCQ controller supports the Nios II processor boot sequence with a small block of on-chip memory mapped to the EPCS/EPCQ controller base address. During Intel Quartus Prime compilation, the EPCS/EPCQ boot copier is designated as the initial contents of this on-chip memory. When booting from EPCS/EPCQ, the **elf2flash** utility does not include a boot copier in the **.flash**. Instead, it includes the application code, packaged into a boot record. The flash programmer downloads the data, which is read by the EPCS/EPCQ boot copier located in on-chip memory.

For IV-series and earlier devices, after the Nios II processor completes reset, the boot copier executes from the on-chip memory block in the EPCS/EPCQ controller. The boot copier reads the Control Block (CB) header to check if a Programmer Object File (POF) FPGA configuration image is located at the beginning of the EPCS device. If it finds such a file, the boot copier reads the POF data to extract the size of the FPGA configuration image. The boot copier then looks for the software application boot record at the EPCS/EPCQ offset immediately following the last byte of the FPGA configuration image.

For V-series and later devices, after the Nios II processor completes reset, the boot copier executes from the on-chip memory block in the EPCS/EPCQ controller. The boot copier reads the POF header to check if the image is a POF file. The POF header is the first three bytes of the FPGA configuration image generated by the **sof2flash** utility. The boot copier then reads the POF header for the size of the FPGA configuration image and looks for the software application boot record at the EPCS/EPCQ offset immediately following the last byte of the FPGA configuration image.

When a boot record is found, the boot copier reads it and copies the application code to RAM. After copying completes, the boot copier reads the entry point of the application code from the boot record. The boot copier executes the jump to that address, and the application software begins executing.

The source code for both variants of the default boot copier is included with the Nios II Embedded Design Suite (EDS) in the <*install directory*>/<version>/nios2eds/components/altera_nios2/boot_loader_sources directory.

5.3.4. Advanced Boot Copier Example

This section describes an advanced boot copier example. You can build the example to run either out of CFI flash or out of on-chip memory, and to support boot images stored in CFI or EPCS/EPCQ flash devices. The example is written in C and is heavily commented, making it easy to customize. This example includes the following features in addition to those provided by the default boot copier:

- Supports two separate boot images
- Supports status messages using a JTAG UART
- Performs error-checking on the boot image data
- Supports non-word-aligned boot images

5.3.4.1. Driver Initialization

To keep memory requirements low, the advanced boot copier example performs only the minimal driver initialization necessary to support the features of the boot copier itself. By default, the example initializes these drivers:

- System Clock Timer
- JTAG UART
- Processor Interrupt Handler

After the boot copier completes initialization of these drivers, it branches to the main application code in RAM, which performs a full initialization of the system drivers.

If you decide that you do not require these components during boot, the example allows you to disable the initialization of their drivers individually, reducing code size.

5.3.4.2. Printing to the JTAG UART

The boot copier in this example prints information to the JTAG UART peripheral during the boot process. Printing is useful for debugging the boot copier, as well as for monitoring the boot status of your system. By default, the example prints basic information such as a start up message, the addresses in flash memory at which it is searching for boot images, and an indication of the image it ultimately selects to boot. You can add your own print messages to the code easily.

The advanced boot copier example avoids using the printf() library function, for the following reasons:

- The printf() library may cause the boot copier to stall if no host is reading output from the JTAG UART.
- The printf() library can potentially consume large amounts of program memory.

5.3.4.2.1. Preventing Stalls by the JTAG UART

The JTAG UART behaves differently than a traditional UART. A traditional UART typically transmits serial data regardless of whether or not an external host is listening. If no host reads the serial data, the data is lost. The JTAG UART, on the

other hand, writes its transmit data to an output buffer and relies on an external host to read from the buffer to empty it. By default, the JTAG UART driver stalls when the output buffer is full. The driver waits for an external host to read from the output buffer before writing more transmit data. This process prevents the loss of transmit data.

During boot, however, it is possible that no host is connected to the JTAG UART. In this case, no transmit data is read from the JTAG UART output buffer. When the output buffer fills, the printf() function stalls the entire program. This stalling is a problem, because the boot copier must continue bringing up the system regardless of whether an external host has connected to the JTAG UART.

To avoid this problem, the advanced boot copier example implements its own printing routine, called my_jtag_write(). This routine includes a user-adjustable timeout feature that allows the JTAG UART to stall the program for a limited timeout period. After the timeout period expires, the program continues without printing any more output to the JTAG UART. Using this routine instead of printf() prevents the boot copier from stalling if no host is connected to the JTAG UART.

5.3.4.2.2. Reducing Memory Use for Printing

The advanced boot copier example also allows you to disable JTAG UART printing altogether. This can significantly reduce the memory requirements of the boot copier. To disable JTAG UART printing in the example, follow these steps:

1. Locate the following line in the <*project*>`/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` file:

   ```
   #define USING_JTAG_UART 1
   ```

2. Replace the line with the following line:

   ```
   #define USING_JTAG_UART 0
   ```

5.3.4.3. Boot Images

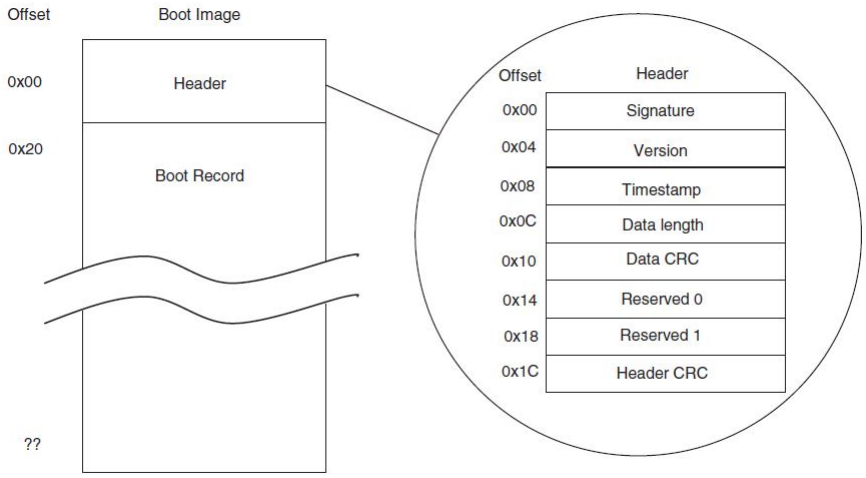
The advanced boot copier example expects to find boot images that conform to a specific format, and supports up to two boot images stored in flash memory. It does not assume the boot image starts at a 32-bit data boundary in flash.

5.3.4.3.1. Boot Image Format

The advanced boot copier example expects to find a boot image that conforms to a specific format. The **make_flash_image_script.sh** script creates boot images that comply with the expected format. The **make_flash_image_script.sh** script runs the **elf2flash** utility to create a boot record of the application from the Executable and Linking Format File (**.elf**), and prepends some header information to that boot record.

Figure 211. Example Boot Image Format

This figure shows the format of a boot image created using the `make_flash_image_script.sh` script.



5.3.4.3.2. Boot Image Header Format

Each boot image includes a header at offset 0x0. The example boot copier uses the header information attached to each boot image to extract information about the image and to make various decisions during the boot process. The **make_flash_image_script.sh** command-shell script automatically adds the header information to any boot image. The following table lists the information contained in the boot image header.

Table 41. Example Boot Image Header Format

Field	Size	Description	Location
Signature	32 bits	The signature used to locate the header in flash memory.	Defined in **make_flash_image_script.sh**. The default boot signature is 0xa5a5a5a5.
Version	32 bits	A binary encoded version identifier for the application.	Defined in **make_flash_image_script.sh**.
Timestamp	32 bits	The time the header was created. Uses the standard C time integer value, seconds since JAN 01, 1970.	Generated by **make_flash_image_script.sh**.
Data length	32 bits	The length of the application data contained in the boot, in bytes.	Generated by **make_flash_image_script.sh**.
Data CRC	32 bits	The CRC32 value for the entire application data	Generated by **make_flash_image_script.sh**.
Unused 0	32 bits	Unspecified purpose.	Defined in **make_flash_image_script.sh**.
Unused 1	32 bits	Unspecified purpose.	Defined in **make_flash_image_script.sh**.
Header CRC	32 bits	The CRC32 value for the header data.	Generated by **make_flash_image_script.sh**.

5.3.4.3.3. Boot Record Format

The boot record immediately follows the boot image header. A boot record is a representation of the application that is loaded by the boot copier. The boot record contains an individual record for each code section of the application. A code section is

a consecutive piece of the code that is linked to a unique region in memory. The boot copier reads the boot record to determine the destination address for each section of the application software code, and performs the appropriate copy operations.

The boot record is necessary because the code sections of a software application might not all be linked to one contiguous region in memory. Often, code sections of an application are scattered all over the memory map. To boot the application, the flash memory must contain the entire application and information about where its parts should be copied in memory. However, the flash memory is too small to contain a copy of the entire memory. The boot record representation packs all the code sections of the application in a single, contiguous block of flash memory.

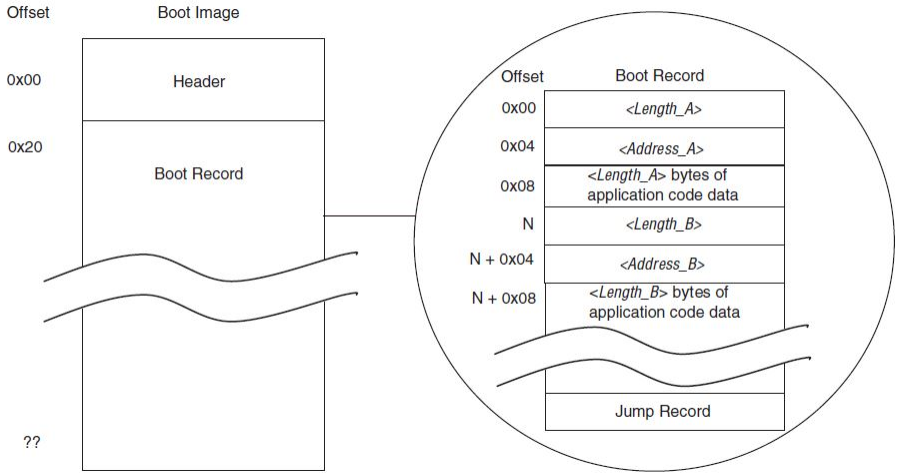
The boot record contains all the code sections of the software application in a contiguous block of data, regardless of where those code sections are linked in RAM. The boot record is a sequence of individual records, each containing the data for a code section, preceded by its destination address and its length. While booting, the boot copier reads the destination address (<*destination address*>) and the length (<*numbytes*>) from the boot record, then copies the following <*numbytes*> bytes from the boot record to <*destination address*>.

The final individual record in the boot record is a special jump record. Reading this record informs the boot copier it has completed copying application code, and that it now needs to jump to the 32-bit address stored in the following four bytes. The 32-bit address is the entry point of the application. Jump records are always encoded as 0x00000000.

The third type of individual record is a halt record. A halt record instructs the boot copier to halt its execution. Halt records are encoded as 0xFFFFFFFF. Erased areas of flash memory contain the value 0xFF in each byte. Therefore, if the boot copier ever encounters an erased area of flash, it interprets it as a halt record and stops execution.

Figure 212. Example Boot Record Memory Map

This figure shows the memory map in an example boot record.



5.3.4.3.4. Choosing a Boot Image

The advanced boot copier example supports up to two boot images stored in flash memory. The boot copier inspects two locations in flash memory, looking for a valid boot image at each location, then chooses one of the images to copy to RAM and execute. The two locations are predesignated as location numbers 1 and 2. To choose a boot image, the boot copier uses the following criteria, in the order in which they appear.

- Image validity
  - If only one valid boot image is found, the boot copier boots using that image.
  - If no valid boot images are found, the boot copier waits five seconds, then jumps back to the Nios II reset address.
- Revision number
  - If both boot images are valid, the boot copier looks at the version number of each image.
  - The boot copier chooses the boot image with the highest version number.
- Timestamp
  - If both boot images have the same version number, the boot copier looks at the timestamp of each image.
  - The boot copier chooses the boot image with the most recent timestamp.
- Default
  - If both boot images have the same timestamp, the boot copier chooses the image in location number 2.

5.3.4.3.5. Word Alignment

In most cases, you can program a Nios II boot image starting at a 32-bit data boundary in flash memory. This placement allows the boot copier to copy application data with 32-bit word transfers. However, the advanced boot copier example does not assume this alignment. If the boot copier finds a valid boot image that is not 32-bit word-aligned in flash memory, the boot copier can still copy the application to RAM accurately. The boot copier uses the memcpy() library function to perform the copying. The memcpy() function requires little memory, and using memcpy() is a fast and robust method for copying data regardless of its alignment in memory.

5.3.4.4. Boot Methods

The advanced boot copier example supports the following boot methods:

- Directly from CFI flash—Boot from CFI flash memory, copy the advanced boot copier from CFI flash memory to RAM, run the advanced boot copier which copies the application from CFI flash memory, and run the application image from RAM.
- From CFI flash, running from on-chip memory—Boot from on-chip RAM, copy the application from CFI flash memory, and run the application image from RAM.
- From EPCS/EPCQ flash, running from on-chip memory—Boot from on-chip RAM, copy the application from EPCS/EPCQ flash memory, and run the application image from RAM.

5.3.4.4.1. Booting Directly From CFI Flash

This method uses a two-stage boot copier. The first boot copier runs from CFI flash and the second runs from RAM. The Nios II reset address is set to an address in CFI flash memory. The default CFI boot copier and advanced boot copier are then programmed at that address in flash. The default CFI boot copier begins executing when the Nios II processor is reset. The default CFI boot copier loads the second boot copier from CFI Flash into RAM and branches to the advanced boot copier's entry point. The advanced boot copier copies the application from CFI flash to RAM, and then branches to the application entry point.

Note: Make sure the code space for the advanced boot copier does not overlap with the application's code space. In this example, we prevent overlap by dividing the SDRAM into two sections, an upper and lower region. The advanced boot copier loads in the upper region of the SDRAM and the application loads in the lower region of the SDRAM.

5.3.4.4.2. Booting From CFI Flash, Running From On-Chip Memory

In this method, the Nios II reset address is set to the base address of a boot ROM implemented as FPGA on-chip memory. The boot copier executable is loaded in the boot ROM when the FPGA is configured, after the hardware design is compiled in the Intel Quartus Prime software. The boot copier begins executing when the Nios II processor is reset. It copies the application code from CFI flash memory to RAM, and then branches to the application entry point.

5.3.4.4.3. Booting From EPCS/EPCQ Flash, Running From On-Chip Memory

This method is very similar to the previous method. The difference is the boot images are stored in EPCS/EPCQ flash, not CFI flash. In this method, set the Nios II reset address to the base address of a boot ROM implemented as FPGA on-chip memory. Load the boot copier executable into the boot ROM when the FPGA is configured, after the hardware design is compiled in the Intel Quartus Prime software. The boot copier begins executing when the Nios II processor is reset, copies the application code from EPCS/EPCQ flash memory to RAM, and then branches to the application entry point.

5.3.4.4.4. Setting the Boot Method

The advanced boot copier example supports all three boot methods described above. The following line in the *<project>*`/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` file controls the method that is implemented:

```
#define BOOT_METHOD <boot method>
```

The options available for *<boot method>* are:

- BOOT_FROM_CFI_FLASH
- BOOT_CFI_FROM_ONCHIP_ROM
- BOOT_EPCS_FROM_ONCHIP_ROM

5.3.4.5. Preventing Overlapping Data in Flash

When you set up your system to boot from a flash memory, consider other data that is also stored in that flash memory. Intel development boards are designed to support storing FPGA configuration images and software boot images together in either type of flash device, CFI or EPCS/EPCQ. When storing multiple images in flash memory, you must ensure that none of the images overlap one another.

5.3.4.5.1. Overlapping Data in CFI Flash

Use the **nios2-elf-size** utility to compute the size of each of your flash images, then choose offsets in flash memory for those images based on their sizes (or estimated future sizes) that ensure they do not overlap.

5.3.4.5.2. Overlapping Data in EPCS/EPCQ Flash

In EPCS/EPCQ flash, the FPGA configuration image must always start at offset 0x0. To avoid programming any boot images on top of the FPGA configuration image, you must determine the end offset of the FPGA configuration image. Convert your FPGA configuration image SRAM Object File (**.sof**) to a **.flash** image using the **sof2flash** utility, then run **nios2-elf-size** on that flash image. The result is the offset at the end of the FPGA configuration image in EPCS/EPCQ flash. Ensure that any software boot images you program into EPCS/EPCQ flash begin at an offset beyond the end of the FPGA configuration image.

5.3.4.6. Boot Copier Code Size

The advanced boot copier example, without modification, compiles to an executable file of size approximately 8500 bytes. If you turn off all the JTAG UART and system clock timer functionality, the example executable size is reduced to approximately 2000 bytes.

By comparison, the code size of the default boot copier, described in the "Default Nios II Boot Copier" section, is approximately 200 bytes when compiled to boot from CFI flash, and approximately 500 bytes when compiled to boot from EPCS/EPCQ flash.

If you require a customizable boot copier that is smaller than 2000 bytes, refer to the "Small Boot Copier Example" section. The small boot copier is written in Nios II assembly language, and includes very few features. When executed, it simply copies a boot record located in CFI flash to RAM, and then branches to the copied application. The compiled code size is approximately 200 bytes.

5.3.5. Implementing the Advanced Boot Copier Example

This section describes the steps required to build and run the advanced boot copier example on the Nios II Ethernet Standard design example.

5.3.5.1. Setting Up the Software Tools and Development Board

To build and run the advanced boot copier example, you must first perform the following steps:

1. Ensure that you have Nios II EDS version 11.0 (or later) and Intel Quartus Prime software version 11.0 (or later) installed on your computer.
2. Connect power and a Intel FPGA Download Cable™ cable to your Intel development board.

5.3.5.2. Creating a Suitable Hardware Design

In the following steps, you can open, modify, and generate a Nios II system on which you can run the advanced boot copier example. You must also decide which boot method you want to implement. Several of the following steps require you to take slightly different actions depending on the boot method you use.

5.3.5.2.1. Opening the Example Project

To open the example project:

1. Verify that you have downloaded and unzipped the file described in the "Hardware Design Files" section to your <*project*> directory on your computer.
2. In the Intel Quartus Prime software, on the File menu, click **Open Project**, and open the <*project*>\niosii_ethernet_standard_ <board> .qpf project file.
   If you intend to boot directly from CFI flash, the standard design example works without additional memory. You can proceed to the "Building the Advanced Boot Copier" section.

5.3.5.2.2. Adding On-chip Boot ROM to the System

To add on-chip boot ROM to the system:

1. On the Tools menu, click **Platform Designer** to start Platform Designer and then select **eth_std_main_system.qsys**. Click **Open** when prompted to open the Platform Designer design file.
2. In Platform Designer, on the **System Contents** tab, expand **Memories and Memory Controllers**, expand **On-Chip**, and select **On-Chip Memory (RAM or ROM)**.
3. Click **Add** to add the component to the system. Use the following settings in specifying the memory:
   - **Memory Type:** RAM (Writable) (not ROM (Read-only))
   - **Data width**: 32 bits
   - **Total memory size**: 16384 bytes

   The specified peripheral size ensures that it can hold the entire code image for the largest version of the example boot copier. This image includes the following code:
   - Resetcode in the `.entry` section
   - The `crt0.s` startup code
   - The `.text` section containing the alt_main entry point
   - The `.rodata` section holding any initialized read only data
   - The `.rwdata` section holding any initialized read/write data
   - The `.bss` section holding initialized and static variables
   - The exception handler located in the `.exception` section

   Some of these sections are copied to the exception RAM—the RAM that contains the exception vectors—when the `crt0.s` startup code executes, but all of the sections are stored initially in this on-chip memory.

4. In the Platform Designer connection matrix, ensure that the slave port of the on-chip memory is connected to the Nios II instruction master and to the Nios II data master, and that the reset port of the on-chip memory is connected to the reset output clock source.
5. If Platform Designer reports an error in the bottom window caused by the address of the new on-chip memory overlapping another peripheral, select a suitable base address for the on-chip memory that does not overlap anything else.
6. Modify the clock entry for the new on-chip memory to ensure that this memory is clocked by the same clock as the **cpu** component.
7. Right-click the new **On-Chip Memory** component, and click **Rename**. Rename the component with a descriptive name such as boot_rom.
8. To enable running the boot copier from on-chip memory, right-click the **cpu** component in your system and click **Edit**.
9. In the Nios II Processor settings window, set the **Reset Vector Memory** to **boot_rom.s1** with an **Offset** of `0x00000000`.
10. Click **Finish** to exit the Nios II Processor settings window.
11. Click **Generate** to generate the Platform Designer system.

5.3.5.3. Building the Advanced Boot Copier

To build the example advanced boot copier in a new Intel Quartus Prime project directory, perform the following steps:

1. Verify that you have downloaded and unzipped the file described in "Software Files" on page 2 to your <*project*> directory on your computer.
2. Open the file <*project*>`/boot_copier_sw/bsp/advanced_boot_copier_bsp/ bootcopier_bsp_settings.tcl` in a text editor.
3. To ensure the proper bsp settings and memory linker locations, edit the following settings in **bootcopier_bsp_settings.tcl**:
   a. Set `ONCHIP` to 1 if you are booting from on-chip memory, or 0 otherwise.
   b. Edit `EXCEPTION_OFFSET` and `RESET_OFFSET` to match values of the CPU component in Platform Designer.
   c. Update `SDRAM_SIZE` and `FLASH_SIZE` to match the memory sizes in Platform Designer.
4. Open the file *project*`/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` in a text editor.
5. Edit the line:

```
#define BOOT_METHOD <boot_method>
```

   to indicate the boot method you intend to use. The following options are available for `<boot_method>`:

   - `BOOT_FROM_CFI_FLASH`
   - `BOOT_CFI_FROM_ONCHIP_ROM`
   - `BOOT_EPCS_FROM_ONCHIP_ROM`

This #define directs the Compiler to build the boot copier appropriately for the boot method you are using.

6. To prevent the application from printing messages to the JTAG UART during boot, edit the line:

```
#define USING_JTAG_UART 1
```

to read:

```
#define USING_JTAG_UART 0
```

This #define directs the Compiler to build the boot copier leaving out all JTAG UART code.

7. Open a Nios II command shell. (On Windows, click **Start ➤ All Programs ➤ Intel FPGA ➤ Nios II EDS ➤ Nios II Command Shell**)
8. Change to the directory <*project*> **/boot_copier_sw/app/advanced_boot_copier**.
9. To create and build the BSP and application projects, type the following command:

```
./create-this-app

//Press ENTER
```

You now have an executable boot copier that is ready to run on the Nios II processor. Next, you must create an application to boot using the new boot copier.

5.3.5.4. Building a Test Application to Boot

To build a test application to boot with the *advanced boot copier* and the *small boot copier*, perform the following steps:

1. Open a Nios II command shell. (On Windows, click **Start ➤ All Programs ➤ Intel FPGA ➤ Nios II EDS ➤ Nios II Command Shell)**.
2. Change to the directory <*project*>`/boot_copier_sw/app/hello_world`.
3. To create and build the test application BSP and application projects, and generate an executable **hello_world.elf** file, type the following command:

```
./create-this-app

//Press ENTER
```

5.3.5.5. Packing the Test Application in a Boot Record

In this section, package the test application to boot in a boot record that the boot copier can understand. To package the application, run a script from a Nios II command shell. The following scripts are included with the design files:

- `make_flash_image_script.sh`
- `make_header.pl`
- `read_flash_image.pl`

To package the test application to boot using the advanced boot copier, perform the following steps:

1. Open <*project*>/boot_copier_sw/app/hello_world/make_flash_image_script.sh in a text editor and update the flash_base and flash_end parameters to match your system.
2. In your Nios II command shell, run the **make_flash_image_script.sh** script to package the **.elf** file in a boot record, by typing the following command:

```
./make_flash_image_script.sh hello_world.elf

//Press ENTER
```

Note: Running this script might issue a warning about an empty loadable segment and display the name of an intermediate file **fake_flash_copier.srec**. You can safely ignore these messages.

The script creates the files **hello_world.elf.flash.bin** and **hello_world.elf.flash.srec** in the current directory. You now have all the binary images needed to boot a test application with the example boot copier. Next, you program these images in the appropriate locations.

5.3.5.6. Booting Directly From CFI Flash Memory

To use the Nios II flash programmer to program the boot copier and the test application in CFI flash memory, perform the following steps:

Note: If you intend to boot from on-chip memory, this section does not apply. Skip ahead to "Booting CFI or EPCS/EPCQ Flash From On-Chip Memory" on page 16.

1. In the Intel Quartus Prime software, on the Tools menu, click **Programmer**.
2. Make sure the <*project*>\niosii_ethernet_standard_<board>.sof filename appears in the **File** column.
3. Make sure the **Program/Configure** option is turned on.
4. Click **Start** to configure your FPGA with the **.sof** file.
5. In a Nios II command shell, change to the directory: <*project*>/boot_copier_sw/app/hello_world.
6. Set the offset in flash memory at which to locate the hello_world boot image, by typing the command:

```
bin2flash --input=hello_world.elf.flash.bin \
--output=hello_world.flash \
--location=0x00240000

//Press ENTER
```

Set the offset to 0x00240000 or 0x00440000, because in boot from CFI flash mode, these are the two locations where the boot copier expects boot images 1 and 2, respectively. The two addresses work equally well.

You can also change these default locations by editing the #define statements for BOOT_IMAGE_1_OFFSET and BOOT_IMAGE_2_OFFSET in the <*project*>/boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c file, and then rebuilding the boot copier.

7. Program the hello_world boot image in flash memory by typing the following command:

```
nios2-flash-programmer --base=<flash_base>\
hello_world.flash

//Press ENTER
```

, where <*flash_base*> is the base address of the CFI flash component in your Platform Designer system.

8. In a Nios II command shell, change to the directory: <*project*>/`boot_copier_sw/app/advanced_boot_copier`.
9. Create the flash memory file for the boot copier by typing the following command:

```
make flash

//Press ENTER
```

This command creates the file <*flash_component*>**.flash**, where <*flash_component*> is the name of the CFI flash component in your Platform Designer system.

10. Program the boot copier to flash memory by typing the following command:

```
nios2-flash-programmer --base=<flash_base> \
<flash_component>.flash

//Press ENTER
```

11. Skip ahead to the "Running the Advanced Boot Copier Example" section.

5.3.5.7. Booting CFI or EPCS/EPCQ Flash From On-Chip Memory

In this section, use the Intel Quartus Prime software to program the boot copier in the **boot_rom** memory of the FPGA, and then use the Nios II flash programmer to program the test application boot record in either CFI or EPCS/EPCQ flash memory.

Note: If you intend to boot directly from CFI flash memory, this section does not apply. Booting directly from CFI flash memory is covered in the "Booting Directly From CFI Flash Memory" section.

To program the boot copier in the FPGA's **boot_rom** memory, perform the following steps:

1. In a Nios II command shell, change to the subdirectory
2. To generate the memory initialization file for the **boot_rom** memory that contains the boot copier, type the following command:

```
elf2hex advanced_boot_copier.elf <boot_rom_start_address>
<boot_rom_end_address> --width=32 --little-endian-mem --create- lanes=0
../../../eth_std_main_system_boot_rom.hex

//Press ENTER
```

3. On the Intel Quartus Prime Processing menu, click **Start Compilation** to compile the project.
4. When compilation is complete, on the Tools menu, click **Programmer**.

5. Make sure the <*project*>\niosii_ethernet_standard_<board> .sof filename appears in the **File** column.
6. Make sure the **Program/Configure** option is turned on.
7. Click **Start** to configure your FPGA with the **.sof** file.

   The **boot_rom** memory on the FPGA now contains an executable image of the example boot copier.

To program the test application in flash memory, perform the following steps:

1. In a Nios II command shell, change to the subdirectory: <*project*> / boot_copier_sw/app/hello_world of your Intel Quartus Prime project directory.
2. Set the offset in flash memory at which to locate the hello_world boot image, by typing one of the following commands:
   a. If you are booting from CFI flash memory, type the following command:

```
bin2flash --input=hello_world.elf.flash.bin \
--output=hello_world.flash \
--location=0x00240000

//Press ENTER
```

   b. If you are booting from EPCS/EPCQ device, type the following command:

```
bin2flash --input=hello_world.elf.flash.bin \
--output=hello_world.flash \
--location=0x00060000

//Press ENTER
```

   Set the offset to 0x00240000 or 0x00440000 when booting from CFI flash memory, and to 0x00060000 or 0x00080000 when booting from an EPCS/EPCQ device, because in boot from the respective flash memory, these are the two locations where the boot copier expects boot images 1 and 2, respectively. In both cases, the two addresses work equally well.

   You can also change these default locations by editing the #define statements for BOOT_IMAGE_1_OFFSET and BOOT_IMAGE_2_OFFSET in the <*project*>/ boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c file, and then rebuilding the boot copier.

   Note: If you edited the flash image offsets in **advanced_boot_copier.c**, specify the --location value as one of the image offsets you defined in **advanced_boot_copier.c**, not the offsets mentioned here.
3. Program the hello_world boot image in flash memory by typing the one of the following commands:
   a. If you are booting from CFI flash memory, type the following command:

```
nios2-flash-programmer --base=<flash_base> \
hello_world.flash

//Press ENTER
```

b. If you are booting from EPCS/EPCQ device, type the following command:

```
nios2-flash-programmer --base=<flash_base> \
--epcs hello_world.flash

//Press ENTER
```

where <*flash_base*> is the base address of the CFI or EPCS/EPCQ flash component in your Platform Designer system.

5.3.5.8. Running the Advanced Boot Copier Example

To run the advanced boot copier example on your development board, perform the following step:

- After the flash programmer completes, in a Nios II command shell, type the following command to reset the Nios II processor:

```
nios2-download -r -g

//Press ENTER
```

The boot loader and the test application both print status messages to the JTAG UART if it is enabled. If the JTAG UART and SYS_CLK_TIMER are initialized in the **bsp/alt_sys_init.c** file, and USING_JTAG_UART remains at value 1 in the <*project*>/`boot_copier_sw/app/advanced_boot_copier/advanced_boot_copier.c` file, you can view these messages.

To see the messages, perform the following step:

- In a Nios II command shell, run the **nios2-terminal** utility by typing the following command:

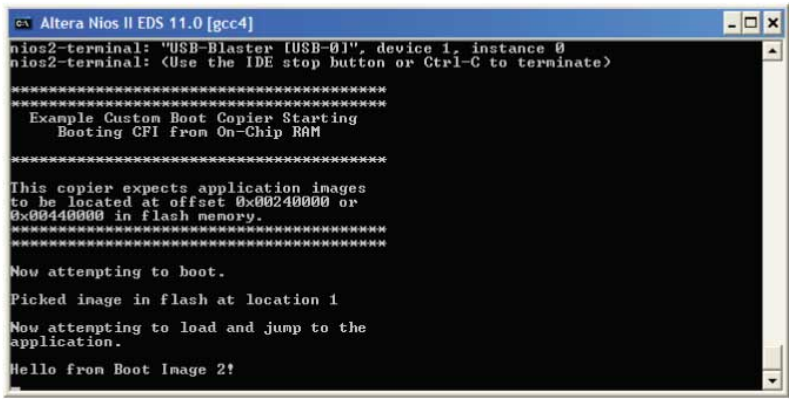
```
nios2-terminal

//Press ENTER
```

Note: If **nios2-terminal** cannot connect to the JTAG UART with the default settings, run it with the --help option for a listing of the command line switches that might be needed.

Note: If your **nios2-terminal** displays truncated output from the boot copier, followed by the boot image output, press the CPU Reset button on your development board to repeat the boot process and view the full output.

Figure 213. Advanced Boot Copier Output

The expected output if you boot CFI flash memory from on-chip RAM.



If the boot copier runs successfully, you see output from **nios2-terminal**, as shown in the figure above.

Note: Your output differs slightly if booting from external memory or booting EPCS/EPCQ flash.

5.3.6. Small Boot Copier Example

This section describes a small code size boot copier example for users interested in using as little memory as possible.

5.3.6.1. Small Boot Copier Features

The small boot copier example is a minimal program designed to use very little program memory. It performs only the following operations:

1. Reads an application boot record from flash memory
2. Copies it to RAM
3. Jumps to the application's entry point

The small boot copier supports only one boot image, does not perform any error checking, and does not support printing messages to the JTAG UART. If you are interested in a more advanced boot copier, refer to the “Advanced Boot Copier Example” section.

Note: This example is designed to run on the Intel Nios II Ethernet Standard design example. For more information about the design example and supported Intel development boards, refer to the *Nios II Ethernet Standard Design Example* located on the "Design Examples" web page on the Intel FPGA website.

5.3.6.2. Implementation in Nios II Assembly Language

To keep the code size as small as possible, write the small boot copier example in Nios II assembly language. All the variables that the boot copier uses are implemented in Nios II processor general purpose registers, not in RAM. Therefore, the boot copier itself has no data memory requirement. The small boot copier has

no **.rodata**, **.rwdata**, stack, or heap section. Because it does not require data memory, this boot copier can easily be relocated anywhere in memory and can even run directly from non-volatile flash memory without setting up a data memory section.

5.3.6.3. System Initialization

The small boot copier performs only the minimum necessary system initialization. The following initialization tasks are performed by the boot copier:

- Clears the processor status register to disable interrupts
- Flushes the instruction cache
- Flushes the processor pipeline

5.3.6.4. Code Size

The small boot copier compiles to an executable file that is only 200 bytes long. This boot copier is small enough to fit in one M9K block, the smallest unit of memory in a Cyclone III FPGA.

5.3.7. Implementing the Small Boot Copier Example

This section describes the steps required to build and run the small boot copier example on the Nios II Ethernet Standard design example. This boot copier is a

bare-minimum, small-code-size version written in assembly language. If you want to build a more full-featured boot copier, refer to the "Implementing the Advanced Boot Copier Example" section.

The small boot copier example is built in a Nios II command shell using the **make**

utility.

5.3.7.1. Setting Up the Software Tools and Development Board

To build and run the small boot copier example, you must first perform the following steps:

1. Ensure that you have Nios II EDS version 11.0 (or later) and the Intel Quartus Prime software version 11.0 (or later) installed on your computer.
2. Connect power and a USB Blaster to your Intel development board.

5.3.7.2. Creating a Suitable Hardware Design

In the following steps, you open, modify, and generate a Nios II system on which you can run the small boot copier example.

To open the example project and add on-chip ROM to the system:

1. Verify that you have downloaded and unzipped the file described in "Hardware Design Files" on page 2 to your <*project*> directory on your computer.
2. In the Intel Quartus Prime software, on the File menu, click **Open Project**, and open the <*project*>\niosii_ethernet_standard_<board> **.qpf** project file.
3. On the Tools menu, click **Platform Designer** to start Platform Designer.

4. In Platform Designer, on the **System Contents** tab, expand **Memories and Memory Controllers**, and select **On-Chip Memory (RAM or ROM)**.
5. Click **Add** to add the component to the system. Specify the following memory settings:
   a. **Memory Type: ROM (Read-only)**
   b. **Data width**: **32** bits
   c. **Total memory size: 512 Bytes**

   Note: The specified on-chip memory size ensures that no memory space is wasted. The smallest usable block of memory in a Cyclone III FPGA is 512 bytes (one M9K block). Although the small boot copier example requires only 200 bytes of memory, the remainder of the M9K block can be used only after you enable it. Therefore, Intel recommends that you enable the entire block, rather than waste it.
6. Right-click the new **On-Chip Memory** and click **Rename**. Specify a descriptive name such as boot_rom.
7. In the Platform Designer connection matrix, ensure that the slave port of the on-chip memory is connected to the Nios II instruction master and to the Nios II data master, and that the reset port of the on-chip memory is connected to the reset output of the clock source.
8. If Platform Designer reports an error in the bottom window caused by the address of the new on-chip memory overlapping another peripheral, select a suitable base address for the on-chip memory that does not overlap anything else.
9. Modify the clock entry for the new on-chip memory to ensure that this memory is clocked by the same clock as the **cpu** component.
10. To enable running the boot copier from on-chip memory, right-click the **cpu** component in your system and click **Edit**.
11. In the Nios II Processor settings window, set the **Reset Vector Memory** to **boot_rom.s1** with an offset of 0x00000000.
12. Click **Finish** to exit the Nios II Processor settings window.
13. Click **Generate** to generate the Platform Designer system.

5.3.7.3. Building the Small Boot Copier Using 'make'

This section describes how to build the example small boot copier from the Nios II command shell.

To build the example small boot copier in a new Intel Quartus Prime project directory, perform the following steps:

1. Verify that you have downloaded and unzipped the file described in "Software Files" on page 2 to your <*project*> directory on your computer.
2. Open a Nios II command shell. (On Windows, click **Start ➤ All Programs ➤ Intel ➤ Nios II EDS ➤ Nios II Command Shell)**.
3. Change to the directory <*project*> **/boot_copier_sw/app/small_boot_copier**.
4. In Platform Designer, determine the base address of your **ext_flash** component (<*flash_base_address*>).
5. In the Nios II command shell, type the following command:

```
make all FLASH_BASE=<flash_base_address> \
BOOT_IMAGE_OFFSET=0x00240000
```

This command builds the small boot copier, hardcoding it to look for a boot image at offset 0x00240000 in flash memory and assumes no other important data is located there. You can freely modify this offset to a value more relevant to your application, but when you program the boot image in flash memory (in step 2 on page 23), ensure that you program it to the same offset you choose in the current step.

Note: This example uses a makefile in place of the Nios II software build tools because you are only compiling a single assembly file, with no associated drivers.

You now have an executable boot copier named **small_boot_copier.hex** that is ready to run on the Nios II processor. Next, you must create an application to boot using the new boot copier.

5.3.7.4. Building a Test Application to Boot

To build a test application to boot with the *advanced boot copier* and the *small boot copier*, perform the following steps:

1. Open a Nios II command shell. (On Windows, click **Start ➤ All Programs ➤ Intel FPGA ➤ Nios II EDS ➤ Nios II Command Shell)**.
2. Change to the directory <*project*>`/boot_copier_sw/app/hello_world`.
3. To create and build the test application BSP and application projects, and generate an executable **hello_world.elf** file, type the following command:

```
./create-this-app

//Press ENTER
```

5.3.7.5. Booting From On-Chip Memory

In this section, use the Intel Quartus Prime software to program the small boot copier in the **boot_rom** memory of the FPGA, and then use the Nios II flash programmer to program the test application boot record in CFI flash memory.

To program the boot copier in the FPGA's **boot_rom** memory, perform the following steps:

1. Change to the <*project*> **/boot_copier_sw/app/small_boot_copier** directory.
2. Copy **small_boot_copier.hex** to the Intel Quartus Prime project directory and rename it **eth_std_main_system_boot_rom.hex** using the following command:

```
cp small_boot_copier.hex ../../../eth_std_main_system_boot_rom.hex

//Press ENTER
```

   Note: You may see a warning that a file by that name already exists in that directory. If you are asked to replace the old file, click **Yes**.

   The next Intel Quartus Prime compilation implements the boot copier executable as the contents of **boot_rom**.
3. If Platform Designer is still open, return to it and click **Exit** to close it.
4. In the Intel Quartus Prime window, on the Assignments menu, click **Settings**.
5. In the **Category** list, click **Compilation Process Settings**, then turn on **Use Smart Compilation**. This option prevents recompilation of the entire design when only an update to the on-chip memory contents is required. The first Intel Quartus Prime compile, however, must be a full compile, because adding an on-chip memory to the system changed the design.
6. On the Processing menu, click **Start Compilation** to compile the Intel Quartus Prime project.
7. When compilation is complete, on the Tools menu, click **Programmer**.
8. Make sure the <*project*> **\niosii_ethernet_standard_** <*board*> **.sof** filename appears in the **File** column.
9. Make sure the **Program/Configure** option is turned on.
10. Click **Start** to configure your FPGA with the **.sof** file.

The **boot_rom** memory on the FPGA now contains an executable image of the example boot copier.

To program the test application in CFI flash memory, perform the following steps:

1. In a Nios II command shell, change to the directory <*project*> **/ boot_copier_sw/app/hello_world**.
2. Set the offset in flash memory at which to locate the hello_world boot image, by typing the command:

```
bin2flash --input=hello_world.elf.flash.bin \
44399204953000--output=hello_world.flash \
--location=0x00240000

//Press ENTER
```

   Set the location to 0x240000, because in boot from CFI flash mode, this is the location where the small boot copier expects to find the boot image. The correct value for the `--location` parameter is the value specified in the command in step 5 in the "Building the Small Boot Copier Using 'make'" section.

3. Program the `hello_world` boot image in flash memory by typing the following command:

```
nios2-flash-programmer --base=<flash_base> \
hello_world.flash

//Press ENTER
```

where <*flash_base*> is the base address of the CFI flash component in your Platform Designer system.

5.3.7.6. Running the Small Boot Copier Example

To run the small boot copier example on your development board:

1. After the flash programmer completes, in a Nios II command shell, type the following command to reset the Nios II processor:

```
nios2-download -r -g

//Press ENTER
```

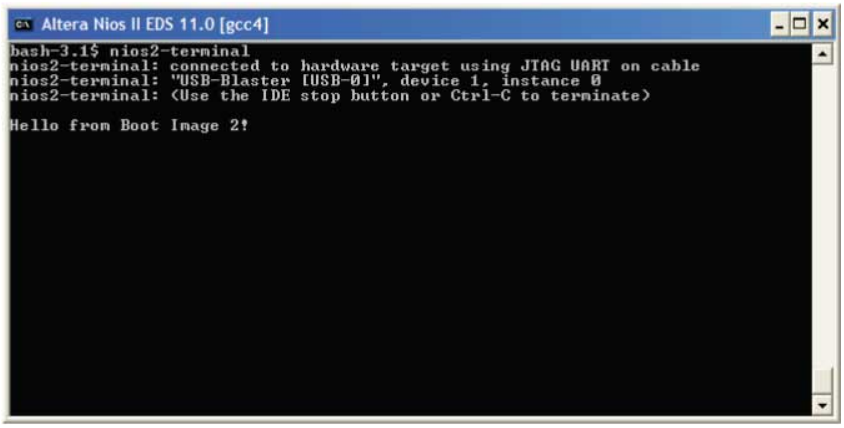
The boot copier should now boot the test application.

2. To test that the test application actually loads and executes, run the **nios2-terminal** utility in the Nios II command shell by typing the following command:

```
nios2-terminal

//Press ENTER
```

If the boot copier runs successfully, you see output from **nios2-terminal**, as shown in the "Small Boot Copier Output" figure.

Figure 214. Small Boot Copier Output



Note: If **nios2-terminal** cannot connect to the JTAG UART with the default settings, run it with the `--help` option for a listing of the command line switches that might be needed.

5.3.8. Debugging Boot Copiers

Some special considerations should be made when attaching the Nios II SBT for Eclipse™ debugger to a processor running boot copier code. The following section discusses the requirements for debugging boot copiers.

5.3.8.1. Hardware and Software Breakpoints

Boot copiers often run from non-volatile memory, which affects the types of breakpoints that can be set in the code. The two types of breakpoints used by the Nios II debugger are software breakpoints and hardware breakpoints. Software breakpoints replace the processor instruction at the breakpoint location with a different instruction that transfers control to the debugger. This replacement method requires that the program memory be writable so that the breakpoint instruction can be written. Because boot copiers often run from non-volatile memory such as flash memory, software breakpoints cannot be set in boot copiers.

Hardware breakpoints detect the address value of the breakpoint on the instruction address bus, and then transfer control to the debugger using hardware. Therefore, a hardware breakpoint can be set in non-volatile memory. Only a hardware breakpoint can be set in a boot copier that runs from flash memory.

5.3.8.2. Enabling Hardware Breakpoints

To enable hardware breakpoints in the Nios II processor:

1. In Platform Designer, open the Nios II MegaWizard interface by double clicking the system's Nios II processor.
2. In the Nios II MegaWizard interface, click the **JTAG Debug Module** page.
3. Select **Level 2** or greater, which allows two simultaneous hardware breakpoints that the Nios II debugger can automatically use.

5.3.8.3. Breaking Before main()

When debugging a boot copier, you may want to start debugging immediately after reset, instead of waiting until reaching the function main(). Some boot copiers do not contain a function main() at all. In these cases, instruct the debugger to set a breakpoint at the program entry point.

5.3.8.4. Setting Up the Debugger

To configure the Nios II debugger for debugging a boot copier:

1. Import your boot copier project to the Nios II SBT for Eclipse by performing the following steps:
   a. Open the Nios II SBT for Eclipse (selecting the workspace folder of your choice).
   b. On the File menu, click **Import**. The **Import** dialog box appears.
   c. Expand the **Nios II Software Build Tools Project** folder, and select **Import Nios II Software Build Tools Project**.
   d. Click **Next**.
   e. Under **Project location**, browse to your boot copier project folder.

f. Enter a project name.

g. Turn off **Clean project when importing** and **Managed project**.

h. Click **Finish**.

2. In the Nios II SBT for Eclipse, highlight the name of the imported bootcopier project, and on the **Run** menu, click **Debug Configurations**.
3. In the **Debug** configuration dialog box, click the **New** icon to create a new debug configuration.
4. Click the **Target Connection** tab.
5. In the **Download** box:
   a. If your boot copier runs from ROM, turn on **Start processor**, turn on **Reset the selected target system**, and turn off **Download ELF to selected target system**.
   b. If your boot copier runs from RAM, turn on **Start processor**, turn off **Reset the selected target system**, and turn on **Download ELF to selected target system**.
6. Click the **Debugger** tab.
7. Turn on **Stop on Startup at:main**.
8. Click **Apply**.
9. Click the **Debug** button to start the debugger. Once connected, the debugger breaks at the entry point of the boot copier.

5.3.9. Externally Controlling the Nios II Boot Process

Another way to boot the Nios II processor is to have a different component, such as another processor, control the boot process externally. In this situation, the external processor reads the Nios II application code from some source and loads it into Nios II program memory. The external processor can retrieve the Nios II application code from various sources. For example, it might read the code from some non-volatile storage medium such as hard disk, or download the code over an Ethernet connection.

The method by which the external processor retrieves the Nios II application code is outside the scope of this document. This section focuses on the process of safely loading the application code in Nios II program memory, then directing the Nios II processor to properly execute the application.

5.3.9.1. Overview

Two different methods are available to implement an externally controlled boot of a Nios II system.

- The external processor unpacks the Nios II boot image and writes the executable application code to Nios II program memory.
- The external processor only copies the boot image to RAM. The Nios II processor takes over from there and unpacks the boot image itself.

The latter method, letting the Nios II processor unpack and load the application from the boot image, is similar to the process of running a normal boot copier on the Nios II processor. The only difference is that instead of a flash programmer placing the boot

image in flash memory, an external processor copies the boot image to RAM. After the external processor releases the Nios II processor from reset, everything happens just as if the Nios II processor were booting from flash memory.

This section focuses on the first method, in which the external processor unpacks the Nios II boot image, copies the application code to Nios II program memory, and then directs the Nios II processor to the application's entry point.

One common requirement, regardless of external boot method, is that you must prevent the Nios II processor from executing any code in the memory space being written by the external processor during the copying and unpacking processes.

Otherwise, you may encounter race condition and data corruption problems. The process described in this section prevents the Nios II processor from executing code by holding it in reset while the application code is copied to Nios II program memory. After the application code is copied, the Nios II processor is released from reset to execute the application.

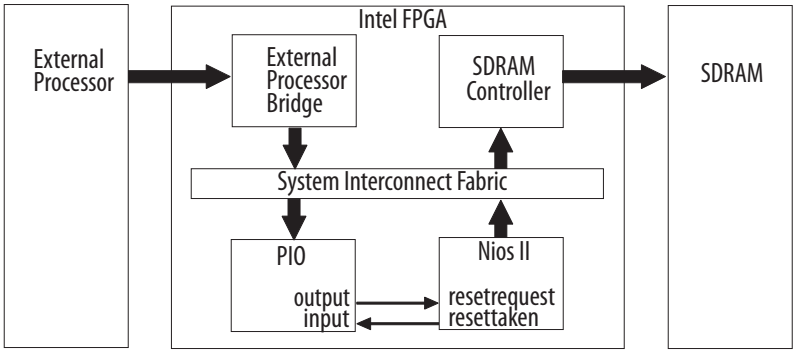
5.3.9.2. Building an Appropriate Platform Designer System

Before you can successfully implement an externally controlled Nios II boot, you must ensure your Platform Designer system contains the necessary hardware. An external processor must be able to access the appropriate system peripherals and control the reset state of the Nios II processor. The following list describes the minimum hardware elements required to support an externally controlled Nios II boot.

- External Processor Bridge
- Nios II processor with the following features:
  - A cpu_resetrequest signal
  - A reset address that points to RAM
  - A one-bit parallel I/O (PIO) peripheral device

Figure 215. Block Diagram of Externally Controlled Nios II Boot System

This block diagram is of a system that can control the boot of a Nios II processor externally.



5.3.9.2.1. External Processor Bridge

To allow an external processor to access peripherals in your Platform Designer system, the system must include a bridge between the Avalon fabric and the external processor bus.

Bridges to external processors can be acquired as intellectual property (IP) or developed internally. Many designers develop their own external processor bridge components for Platform Designer because it is usually relatively straightforward to bridge the Avalon fabric architecture to other bus protocols. The Component Editor tool, available in Platform Designer, is useful for creating IP such as external processor bridges.

Note: For a list of bridge IP available from Intel, refer to the "All Reference Designs" page on the Intel FPGA website.

5.3.9.2.2. The cpu_resetrequest Signal

In versions 6.0 and later of the Nios II processor, an optional cpu_resetrequest signal is available to control the reset state of the processor. This signal differs from the normal Platform Designer system-wide reset signal reset_n—the cpu_resetrequest signal resets the Nios II processor only. The rest of the Platform Designer system remains operational. This signal holds the Nios II processor in reset while code is moved into the Nios II program memory.

The cpu_resetrequest signal does not cause the Nios II processor to enter the reset state immediately. When cpu_resetrequest is held high, the Nios II processor finishes executing any instructions currently in the pipeline, then enters reset. This process may take an indeterminate number of clock cycles, so a status signal cpu_resettaken is driven high by the Nios II processor when it reaches the reset state. The processor holds this signal high for one cycle. The cpu_resettaken signal continues to assert periodically while the cpu_resetrequest signal is held high.

To enable the cpu_resetrequest signal, open a project in Platform Designer that contains a Nios II processor. Double-click the Nios II component to open the Nios II MegaWizard interface, then click the **Advanced Features** page. Turn on **Include cpu_resetrequest** and **cpu_resettaken signals** to enable the signals. They appear as ports on your top-level Platform Designer system after you regenerate the system.

5.3.9.2.3. Nios II Reset Address

The Nios II reset address is the address of the first instruction the processor executes after it is released from reset. Therefore, in a Nios II system capable of externally controlled boot, the Nios II reset address must point to a writable memory (RAM). This class of reset address is typically not what you want in a traditional boot scenario, but in the external boot control situation described in this section, it is important that the Nios II reset address point to RAM.

The Nios II reset address must point to RAM because, to direct the Nios II processor to the application code that was just copied into RAM, the external processor must be able to write the first instruction (or instructions) that the Nios II processor executes upon reset. Typically, the instruction written to the reset address is an unconditional branch (br) to the entry point of the application.

You can choose any unused 32-bit location in RAM as the reset address for the Nios II processor, but the base address (offset 0x0) of the Nios II program memory—the memory region that contains the .text section—is usually a good choice. By default, the Nios II exception table is placed at offset 0x20 in the program memory, and the remainder of the application code is placed following the exception table in consecutive memory. This arrangement leaves offsets 0x0 through 0x1C available. A reset address at offset 0x0 guarantees that the difference between the reset address and the application entry point—assumed to be early in the application code—never

exceeds 64 Kbytes, as required for this process to work. For a description of why the difference cannot exceed 64 Kbytes, see the discussion of instruction step 1 in the "External Boot Flow" section.

5.3.9.2.4. One-Bit PIO Peripheral

A one-bit PIO peripheral is needed to control the Nios II cpu_resetrequest signal from the external processor. The external processor accesses the Avalon-mapped PIO peripheral through the external processor bridge. The external processor writes the value 1 to the PIO to assert the cpu_resetrequest pin, or the value 0 to de-assert it.

The external processor can also read the state of the cpu_resettaken signal using the same PIO peripheral. However, the Nios II processor asserts the cpu_resettaken signal for only one clock cycle at a time. Therefore, sampling this signal from software to see when reset has been achieved does not work. The signal can easily assert and de-assert again between samples, so that a valid assertion of cpu_resettaken by the Nios II processor might never be captured by the external processor.

The PIO component included with Platform Designer includes an edge-capture feature to use in this situation. The edge-capture feature sets a bit in the edge-capture register of the PIO whenever an edge of the predefined type is seen on that bit of the PIO's input port. The external processor can read the edge-capture register any time after it asserts cpu_resetrequest. If the cpu_resettaken signal was asserted any time since the cpu_resetrequest assertion, the relevant bit in the PIO's edge-capture register is set.

To add a PIO component configured to use the edge-capture feature to detect assertions of cpu_resettaken to your system, perform the following steps:

1. Open your system in Platform Designer.
2. On the **System Contents** tab, under **Peripherals**, and then under **Microcontroller Peripherals**, click the **PIO (Parallel I/O)** component.
3. Click **Add**.
4. In the **PIO** MegaWizard interface, set the width to one bit and select **InOut for Directions**.
5. Under the **Edge capture register** section, check the **Synchronously Capture** box, and then select **RISING** for Edge type.
6. Click **Finish** to add the PIO component to your system.

Your system now contains a PIO component capable of asserting the Nios II `cpu_resetrequest` signal and detecting rising edges on the `cpu_resettaken` signal.

Note: Platform Designer does not automatically connect the input and output ports of the PIO component to the Nios II `cpu_resettaken` and `cpu_resetrequest` signals. After Platform Designer generation, you must make these connections at the top level in the Intel Quartus Prime project.

5.3.9.3. The Boot Process

Now that you have learned the important hardware aspects of externally controlling the Nios II boot process, this section describes the entire boot process from the perspective of the software running on the external processor.

5.3.9.3.1. Boot Images

The procedure described here assumes you have a Nios II boot image in the format described in the "Boot Images" section.

5.3.9.3.2. Example C Code

In the directory **boot_copier_src/app/external_boot**, you can find sample C source code that you can run on an external processor to control the boot of a Nios II processor. The code is heavily commented, making it relatively easy to modify and customize. The example code happens to retrieve the boot image from offset 0x0 of a CFI flash, but in a real system, the boot image could come from anywhere. That part of the process is left to your discretion.

5.3.9.3.3. External Boot Flow

The following section describes the boot flow implemented in the example C code mentioned in the previous section. These steps are written from the perspective of software running on an external processor which is responsible for controlling the Nios II boot process.

1. Retrieve the Nios II boot image.

   The software can retrieve the Nios II boot image any number of ways. Common methods include reading the boot image from non-volatile storage such as hard disk or flash memory, downloading it over an Ethernet connection, or passing in a pointer to its location in RAM. Most important is that the image be locally accessible in its entirety before you attempt to unpack it and copy it to Nios II program memory.

2. Hold the Nios II processor in reset using the one-bit PIO, by performing the following actions:

   a. Write any 32-bit value to offset 0x3 of the PIO component to clear the edge-capture register. Using the edge-capture register to detect when the cpu_resettaken signal goes high requires that you clear the edge-captureregister first to ensure the register value does not represent an edge event that occurred in the past.

   b. Write the value 1 to offset 0x0 in the PIO component to assert the Nios II `cpu_resetrequest signal`.

   c. Continuously poll offset 0x3 of the PIO component until it holds the value 1. Thisvalue indicates that the Nios II `cpu_resettaken` signal transitioned high, which ensures the Nios II processor is now in a reset state and you can safely begin copying application code to its program memory.

3. Copy the application to its destination address in memory space.

4. Parse the boot record to copy each section of the application code to its appropriate location in Nios II program memory.

   The final individual record in the boot record is a jump record. Be sure to save the jump value, which contains the entry point of the application code. In the next step, you must direct the Nios II processor to the application entry point.

5. Construct a branch instruction to place at the Nios II reset address.

Constructing a Nios II branch (br) instruction allows Nios II to branch from its reset address to the entry point of the application code. Because the Nios II branch instruction is relative, meaning it branches relative to the current instruction, you need to know both the Nios II reset address, and the application entry point address.

In the example code, the Nios II reset address is simply defined at the top of the file. If your Nios II reset address changes, you must also change the relevant #define in the example code.

6. Subtract the reset address from the entry point address (saved in the previous step) to obtain the offset.

   For Nios II pipelining, the branch instruction actually requires the offset to be relative to the next instruction, so subtract 4 from the offset to obtain the actual offset needed for the instruction.

   Note: Because all Nios II instructions are 32 bits, every address and offset must be a multiple of 4.

   Note: The offset used in the branch instruction is 16 bits, so your reset address must be less than 64Kbytes away from the application entry point in memory space.

   Using the branch instruction encoding shown in the following table, construct a branch instruction from the offset. The following C statements create a properly encoded instruction from the reset address and entry point:

```
int offset = entry_point – reset_address; unsigned int inst = ((offset - 4)
<< 6) | 0x6;
```

Table 42. Nios II Branch Instruction Encoding

31-27	26-22	21-6	5-0
0	0	16-bit offset - 4	0x06

7. Write the branch instruction to the Nios II reset address.
8. Release the Nios II processor from reset.

   Write a zero to offset 0x0 of the PIO peripheral to deassert the Nios II `cpu_resetrequest` signal. The Nios II processor should come out of reset, execute the branch instruction, branch to the entry point of the application, and begin to execute it.

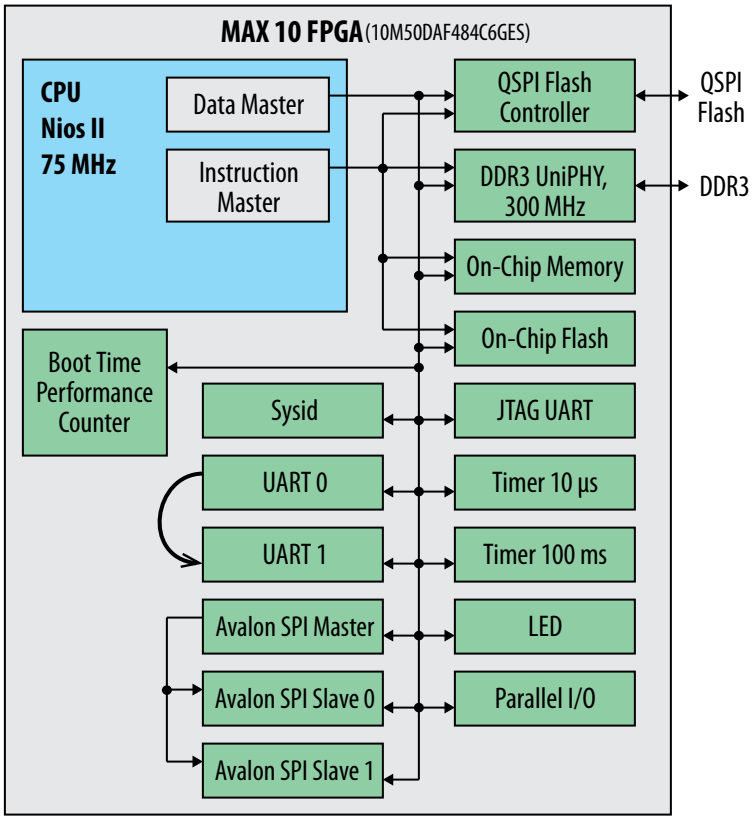
   Booting is now complete. The Nios II processor is off and running, so the external processor can go about its other system tasks.

5.4. Boot Time Performance Analysis

Intel performed a boot time performance analysis for Intel MAX 10 FPGA based on several use cases. This section provides guidance on how to reduce Nios II boot time in MAX10 design and boot time estimation based on performance analysis. You can use the information in this section as a guideline when designing your custom design.

Figure 216. Boot Time Performance Analysis Design Block Diagram

Diagram shows the design was being used to run the Intel MAX 10 FPGA boot time performance analysis. The Nios II processor was configured with special settings for some of the use cases.



Regardless of the boot device, there are only 2 types of boot methods:

1. Nios II processor application execute-in-place.
2. Nios II processor application copied from boot device to RAM using boot copier.

For boot performance analysis, the Nios II processor application sizes varies between 10kB to 64kB.

5.4.1. Boot Time Performance Analysis Design Example

The design example was tested on MAX10 10M50 Development Kit (Rev B) on ACDS 15.0 Build 145.

The design example boots from the On-chip Flash (UFM). If you want to boot from other boot memory device, you have to change the reset vector and BSP settings according to Table 40 on page 349, located in the "Summary of Nios II Processor Vector Configurations and BSP Settings".

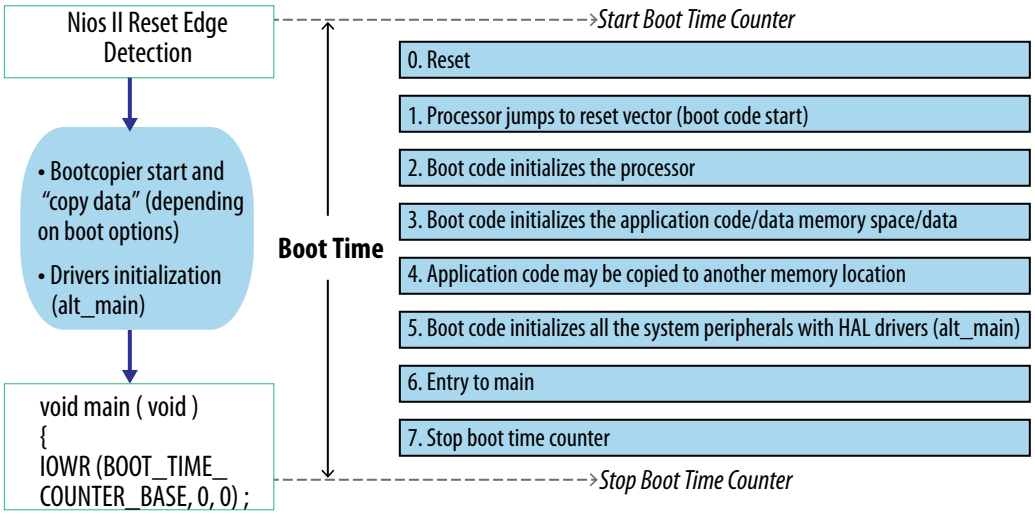
5.4.2. Boot Time Measurement Strategy

The boot time measurement was done using a boot time performance counter. The boot time counter starts when Nios II comes out of reset and the system will start the boot sequence according to the selected boot options:

1. Nios II processor application execute-in-place:
   - The `alt_load ()` function copies data sections (`.rodata`, `.rwdata` , `.exceptions`) from boot memory to RAM.
   - The code section (`.text`) remains in the boot memory.
   - Automatically jump to the user generated application code in the boot memory after copy completed.
   - Run system/ driver initialization (alt_main).
2. Nios II processor application copied from boot device to RAM using boot copier:
   - Bootcopier locates the software application in the boot memory and copies the software application to RAM.
   - Automatically jump to the user generated application code in the RAM after copy completed.
   - Run system/ driver initialization (alt_main).

The boot time counter is controlled through software and it will stop once the driver initialization completes.

Figure 217. Boot Time Measurement



5.4.3. Reducing Nios II Boot Time in Intel MAX 10 FPGA Design

There are a few elements that you can consider to improve the Nios II boot time in MAX10 designs. Based on the Nios II Boot Time analysis that has performed, the following subsections can be referred to for general guidance.

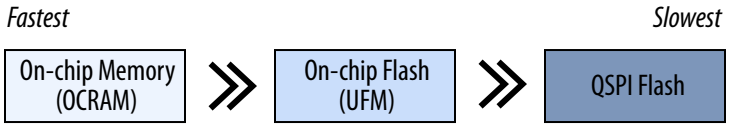
5.4.3.1. Boot Methods

1. Nios II application boot time for execute-in-place is faster than using a boot copy/run from RAM based boot (due to the time taken to copy the code to RAM being much longer than the time gained by the higher performance when running from RAM instead of ROM).
2. The boot copier method is recommended for systems that require higher performance. Although this configuration takes longer to boot, it delivers higher application performance compared to an execute-in-place configuration.

5.4.3.2. Boot Device Performance

1. For Nios II application execute-in-place, different boot devices will have different boot times based on their individual memory performance.
2. The following shows the performance for the supported boot device in Intel MAX 10 FPGAs, when Nios II applications execute-in-place:

Figure 218. Supported Boot Device Performance



5.4.3.3. Peripheral Initialization

Nios II systems initialize all HAL peripherals before main() by default, therefore the boot time has a dependency on the peripherals selected. Peripherals that are slow to initialize or have external dependencies, will increase the boot time and potentially make it less deterministic. If this occurs, you need to calibrate the external memory such as DDR3 for it to work properly.

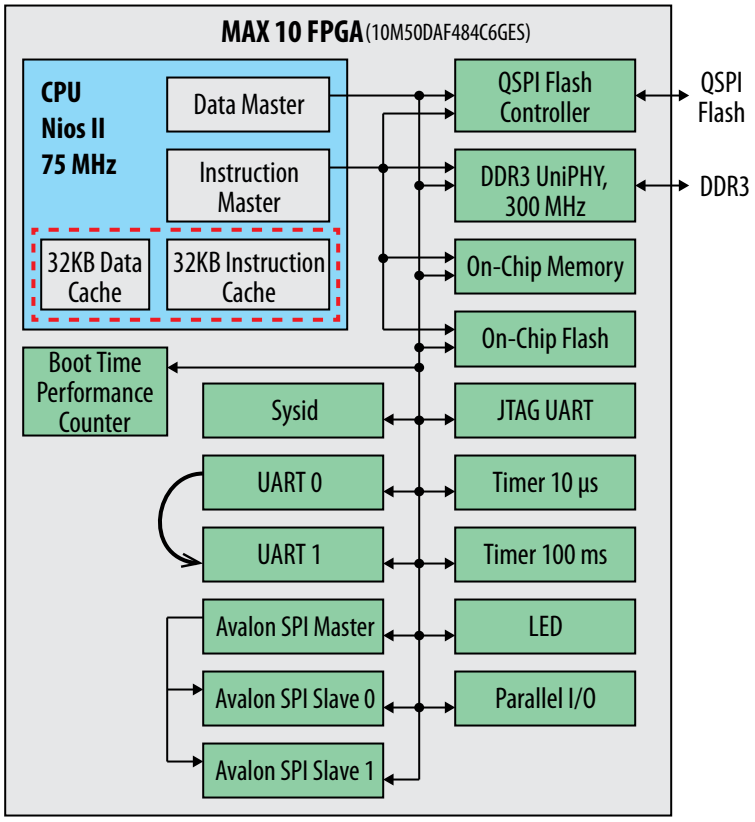
DDR3 is an example of a peripheral where the initialization time is significant in comparison to the boot time. The calibration time is long particularly when compared to boot times for execute-in-place boot configurations. The calibration time will significantly impact Nios II application that execute-in-place.

To avoid this, in execute-in-place boot configurations, remove the external memory from the Nios II linker region if it is not in use. If size is not an issue, you can choose to use OCRAM. If you are confident working with Nios II software, another option is to remove the DDR3 initialization routine from the boot code and initialize the memory later–once the application code has started running.

5.4.3.4. Nios II Processor Caches

1. Enable Nios II processor caches. Caches improve the boot time because the data and instruction caches reduce memory bandwidth limitations during the boot sequence.
2. You can add caches to your Nios II hardware configuration for both execute-in-place or boot copier boot methods to improve boot time.

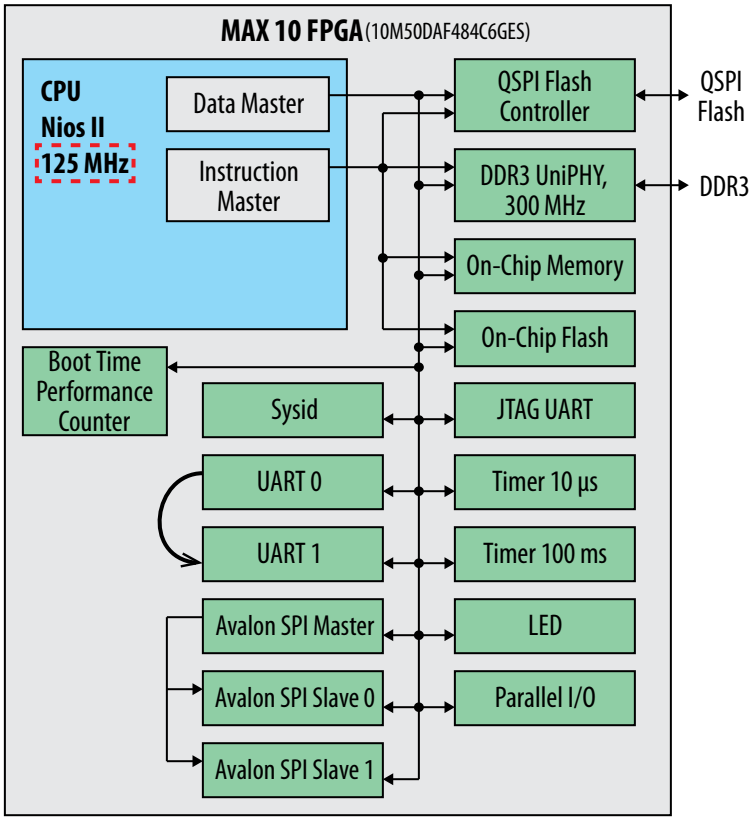
Figure 219. Nios II Design Block Diagram with Processor Caches Enabled



5.4.3.5. System Speed

Using Nios II processor with higher clock speed improves the boot time. For example, you can increase the Nios II processor speed from 75 Mhz to 125 MHz.

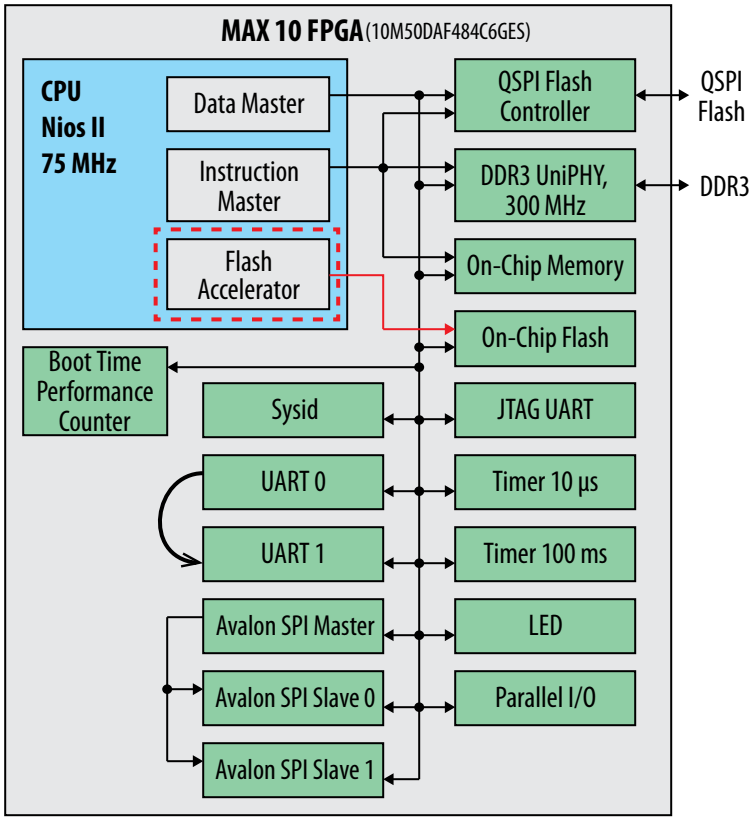
Figure 220. Nios II Design Block Diagram with Higher System Speed



5.4.3.6. Intel MAX 10 FPGA Nios II Flash Accelerator

Enable the Nios II flash accelerator when system resets from the Intel MAX 10 FPGA on-chip flash to improve the boot time.

Figure 221. Nios II Design Block Diagram with Flash Accelerator Enabled



5.4.4. Boot Time Performance and Estimation

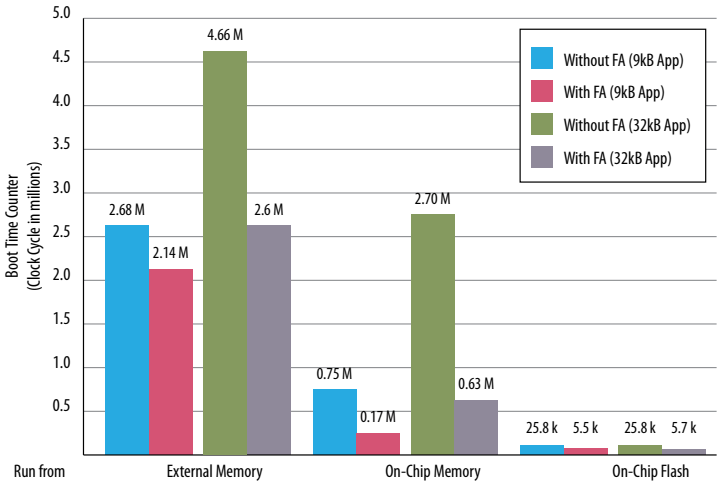
5.4.4.1. Boot Time Performance for Design that Boots from On Chip Flash with Flash Accelerator

The following bar graphs show the boot time performance for designs that boot from on-chip flash (application code stored in on-chip flash) with and without the flash accelerator (FA) unit. Three scenarios have been evaluated:

1. Design with EMIF that runs from external memory DDR3 (using boot copier).
2. Design without EMIF that runs from on-chip memory (using boot copier).
3. Design without EMIF that runs from on-chip flash (execute in-place).

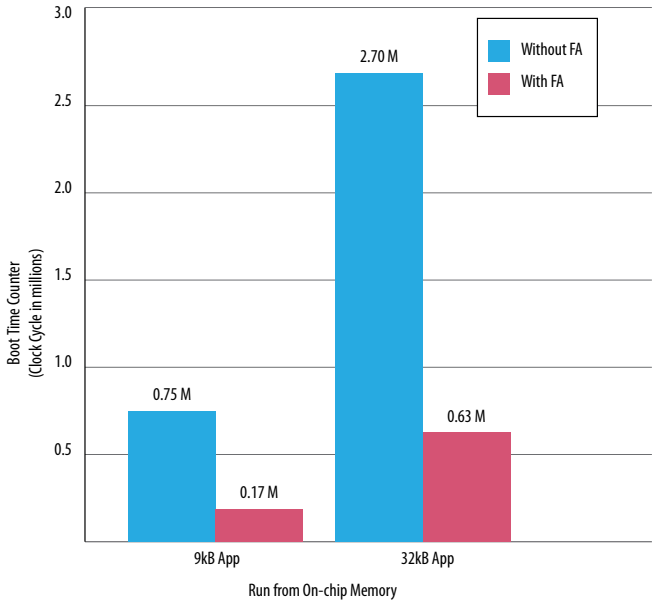
Note: None of the designs include an instruction or data cache.

Figure 222. Summary of Boot Times for Designs that Boot from On-Chip Flash with or without Flash Accelerator



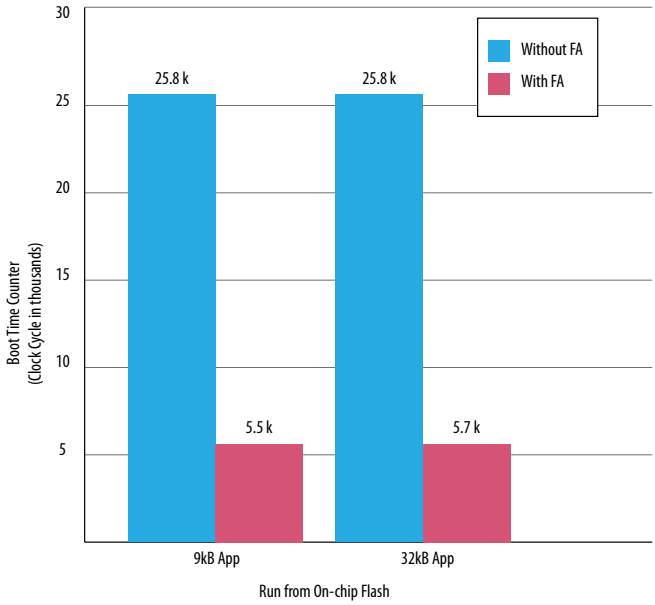
The boot time was reduced by ~20% and ~44% for 9 kB and 32 kB application sizes respectively when the Nios II FA is enabled. Larger application sizes will result in longer boot times because the code has to be copied from the on-chip flash to the external memory during the boot process. Overall, the boot time for the scenario where the boot code is running from the external memory is the slowest. This is due to the long time taken by external memory calibration during system boot up and the time taken to copy the application code to external memory.

Figure 223. Boot Time for Design without EMIF that runs from On-Chip Memory



The boot time was reduced by ~77% and ~76% for 9 kB and 32 kB application sizes respectively when the Nios II FA is enabled. Running the boot code from the on-chip memory is faster than running from external memory because the on-chip memory does not need to go through memory calibration during system boot up. Additionally, access time to the on-chip memory are faster than those for external DDR3 memory.

Figure 224. Boot Time for Design without EMIF that runs from On-Chip Flash (Execute In-Place)



The boot time was reduced by ~78% and ~77% for 9 kB and 32 kB application sizes respectively when the Nios II FA is enabled. Running the boot code from the on-chip flash (execute in-place) is faster than running from external memory and on-chip memory. This is because the code executes in-place directly from the on-chip flash and does not need to be copied into the external memory or on-chip memory for execution which saves a lot of time.

5.4.4.2. Boot Time Estimation

The tables below show estimates of boot time for different Intel MAX 10 FPGA boot configurations. The boot time shown will help you to gauge the boot configuration required for your system design.

Note:

- Instruction or data cache and flash accelerator are not enabled in the design.
- Boot time values are based on design with 75 MHz Nios II processor speed.

Table 43. Boot Time Estimation on Execute-in-Place Test Cases

Test Case	Boot From	Boot Time Counter (Approximate Nios II Clock cycles)(25)	Boot Time (millisecond)
Design without External Memory	On Chip Flash	41,000	0.55
	On Chip Memory	18,000	0.24
	QSPI Flash	410,000	5.5
Design with External Memory	On Chip Flash	2,000,000	27
	On Chip Memory	2,000,000	27
	QSPI Flash	2,300,000	31

Table 44. Boot Time Estimation on Boot Copier Test Cases

Test Case	Boot from	Run From	Boot Time Counter (Clock cycle)(26)	Boot Time (millisecond)
Design without External Memory	On Chip Flash	On Chip Memory	2,800,000	37
	Quad SPI Flash	On Chip Memory	28,000,000	370
Design with External Memory	On Chip Flash	External Memory	4,700,000	63
		On Chip Memory	2,800,000	37
	Quad SPI Flash	External Memory	30,000,000	400
		On Chip Memory	30,000,000	400

(25) Boot time counter is applicable to all software application (.elf) sizes.

(26) Boot time counter is for every 32kB .elf size.

5.5. Nios II Configuration and Booting Solutions Revision History

Document Version	Changes
2020.07.22	Clarified information specific to the Intel MAX 10.
2017.11.06	• Merged the following documents into *Nios II Configuration and Booting Solutions* chapter: — *AN730: Nios II Processor Booting Methods In Intel MAX 10 Devices* — *AN736: Nios II Processor Booting From Intel FPGA Serial Flash (EPCQ)* — *AN458: Alternative Nios II Boot Methods* — *Generic Nios II Booting Methods User Guide* • Updated the information in *Nios II Processor Booting Methods* section for the following: — On-Chip Flash — EPCQ Flash — QSPI Flash — On-Chip Memory (OCRAM) — CFI Flash • Editorial changes.
2016.12.19	Initial release.

Related Information

Document Revision History for Embedded Design Handbook on page 6

6. Nios II Debug, Verification, and Simulation

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.

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

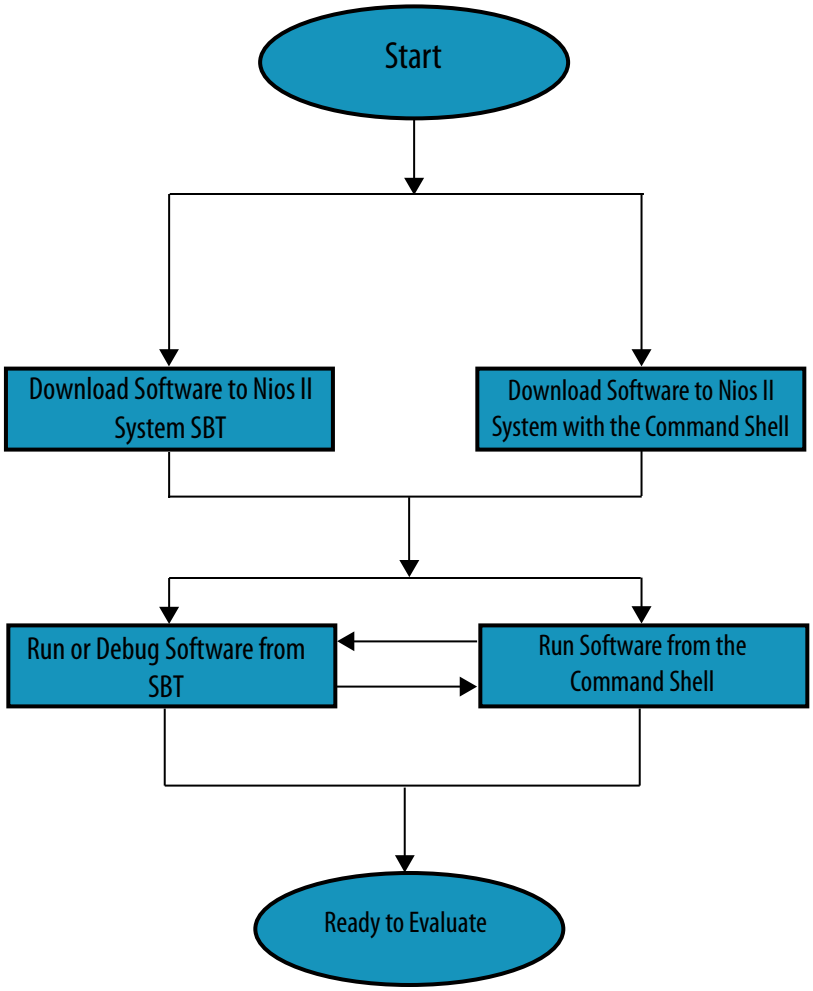
ISO 9001:2015 Registered

6.1. 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 225. 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 Platform Designer 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 Intel Quartus Prime Handbook. Online training is available at the Intel Training page of the Intel 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 395
- Nios II Command-Line Tools on page 120
- Verification and Board Bring-Up on page 418
- Analyzing and Debugging Designs with System Console
- Intel FPGA Training

6.2. 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 395
- Run-Time Analysis Debug Techniques on page 404

6.2.1. 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 395
- Signal Tap II Embedded Logic Analyzer on page 403
- Lauterbach Trace32 Debugger and PowerTrace Hardware on page 403
- Data Display Debuggers on page 404

6.2.1.1. 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.

Platform Designer 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. Platform Designer 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 396
- Project Templates on page 397
- Configuration Options on page 397
- Nios II GDB Console and GDB Commands on page 400

- Nios II Console View and stdio Library Functions on page 400
- Importing Projects Created Using the Nios II Software Build Tools on page 401
- Selecting a Processor Instance in a Multiple Processor Design on page 401

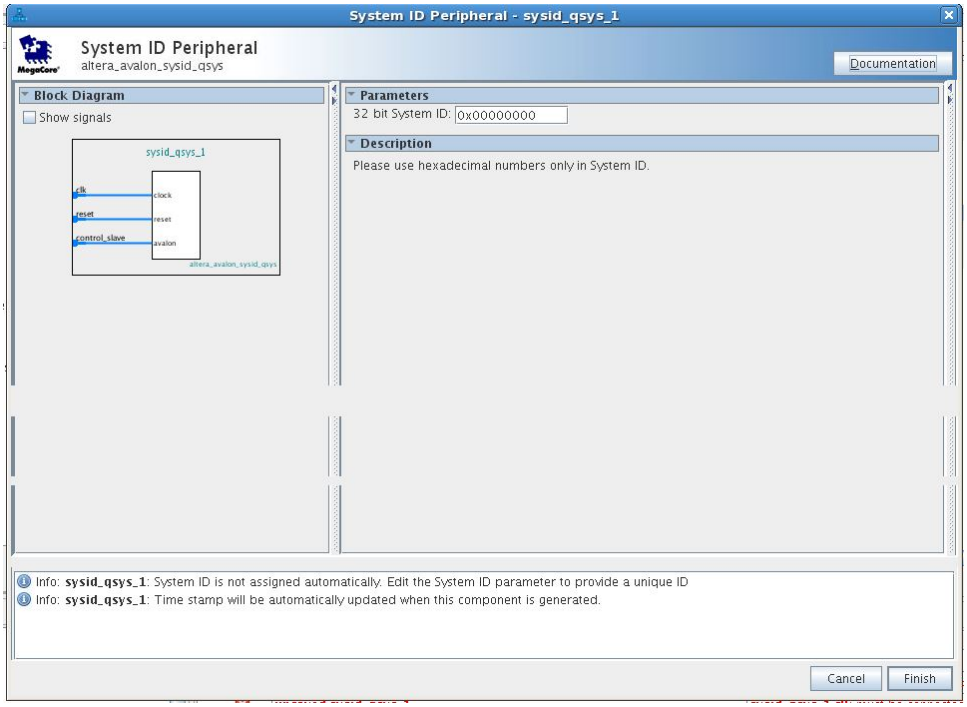
6.2.1.1.1. Nios II System ID

The system identifier (ID) feature is available as a system component in Platform Designer. The component allows the debugger to identify attempts to download software projects with BSP projects that were generated for a different Platform Designer 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 Platform Designer design include a system ID component. In the figure below shows an Platform Designer system with a system ID component.

Figure 226. Platform Designer 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

6.2.1.1.2. 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 Platform Designer 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.

6.2.1.1.3. 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 398
- Show Make Commands on page 399
- Show Line Numbers on page 399

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 31. 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:

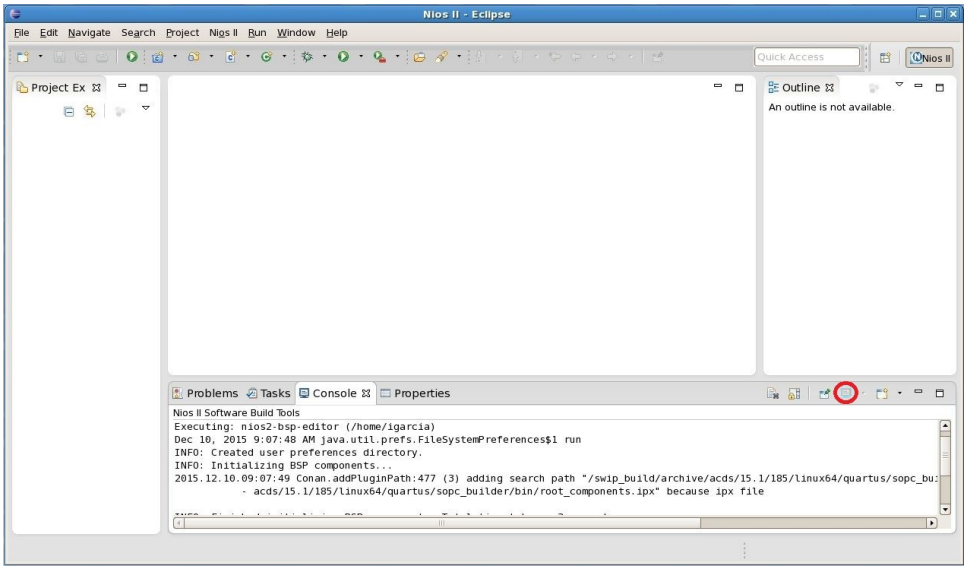
1. On the Window menu, click **Preferences**.
2. On the list to the left, under **General**, under **Editors**, select **Text Editors**.
3. On the **Text Editors** page, turn on **Show line numbers**.
4. Click **Apply**.
5. Click **OK**.

6.2.1.1.4. 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 227. 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.

6.2.1.1.5. 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

6.2.1.1.6. 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

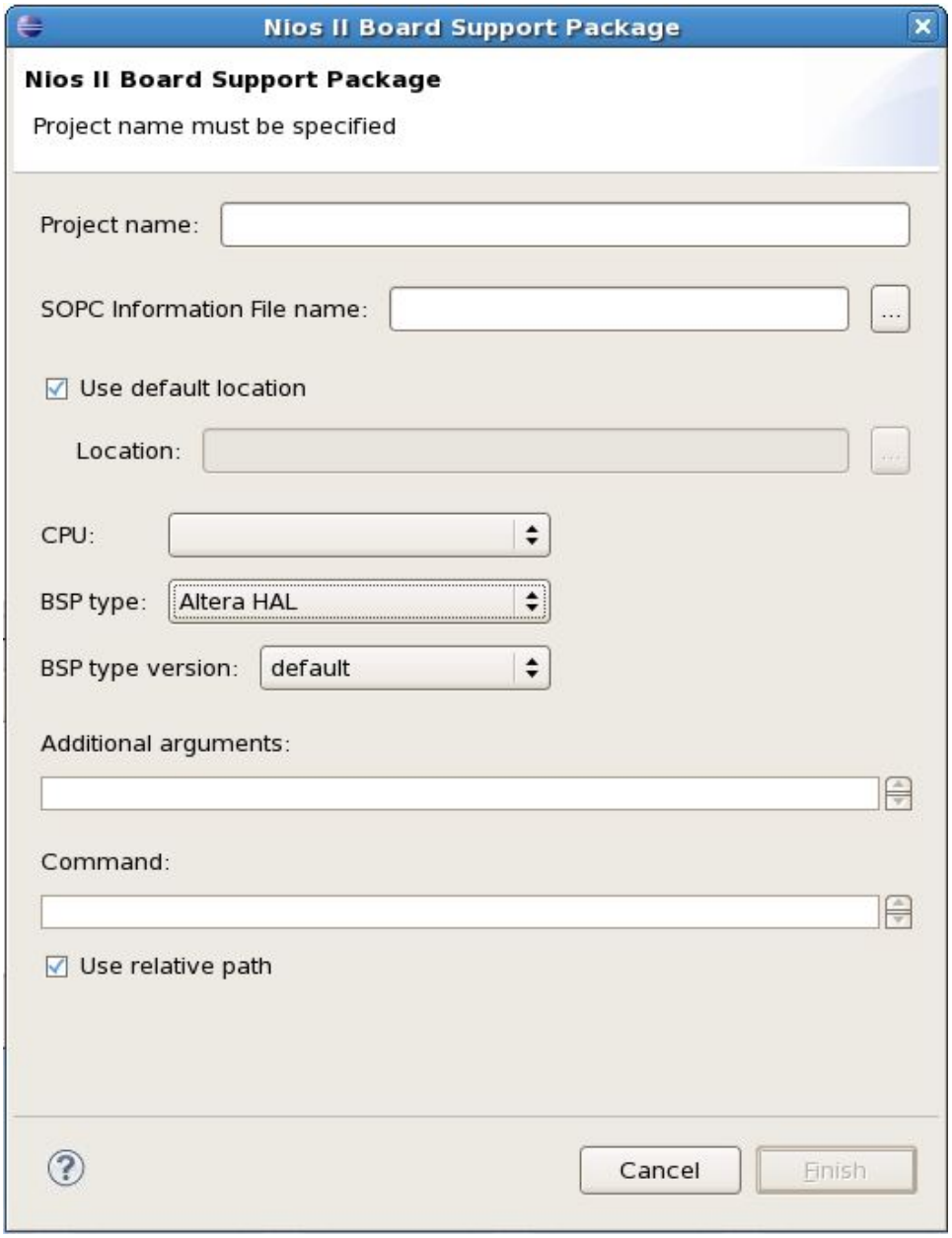
Getting Started with the Graphical User Interface

6.2.1.1.7. 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 228. 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 32. Two-FPGA System Response to jtagconfig Command

```
[Platform Designer]$ jtagconfig -n
1) Intel FPGA Download Cable [USB-0]
120930DD EP2S60
Node 19104600
```

```
Node 0C006E00
2) Intel FPGA Download Cable [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 Intel FPGA Download Cable cables. The cable attached to the USB-0 port is connected to a JTAG node in a Platform Designer subsystem with a single Nios II core. The cable attached to the USB-1 port is connected to a JTAG node in a Platform Designer 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.

6.2.1.2. Signal Tap II Embedded Logic Analyzer

The Signal Tap 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 Signal Tap II embedded logic analyzer, refer to the Design Debugging Using the Signal Tap II Embedded Logic Analyzer chapter in volume 3 of the *Intel Quartus Prime Handbook* and the Verification and Board Bring-Up chapter of this handbook.

The Nios II plug-in for the Signal Tap 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 Signal Tap II embedded logic analyzer, refer to *AN446: Debugging Nios II Systems with the Signal Tap II Logic Analyzer*.

Related Information

- Verification and Board Bring-Up on page 418
- Design Debugging Using the Signal Tap Logic Analyzer
- AN446: Debugging Nios II Systems with the Signal Tap II Logic Analyzer.

6.2.1.3. 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 Intel FPGA USB Download 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

6.2.1.4. 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.

6.2.2. Run-Time Analysis Debug Techniques

This section discusses methods and tools available to analyze a running software system.

6.2.2.1. Software Profiling

Intel 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 Platform Designer 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 Platform Designer 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 Platform Designer 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 Platform Designer 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 137
- AN391: Profiling Nios II Systems
- Interval Timer Core
- Performance Counter Core

6.2.2.2. 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 Platform Designer to debug level 2 or higher. To configure the Nios II processor's debug level in Platform Designer to the appropriate level, perform the following steps:

1. On the Platform Designer 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 229 on page 407
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 229 on page 407 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

6.2.2.3. 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.

6.2.2.3.1. 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

6.2.2.4. Hardware Abstraction Layer (HAL)

The Intel FPGA HAL provides the interfaces and resources required by the device drivers for most Platform Designer system peripherals. You can customize and debug these drivers for your own Platform Designer system.

To learn more about debugging HAL device drivers and Platform Designer 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

6.2.2.5. Breakpoints

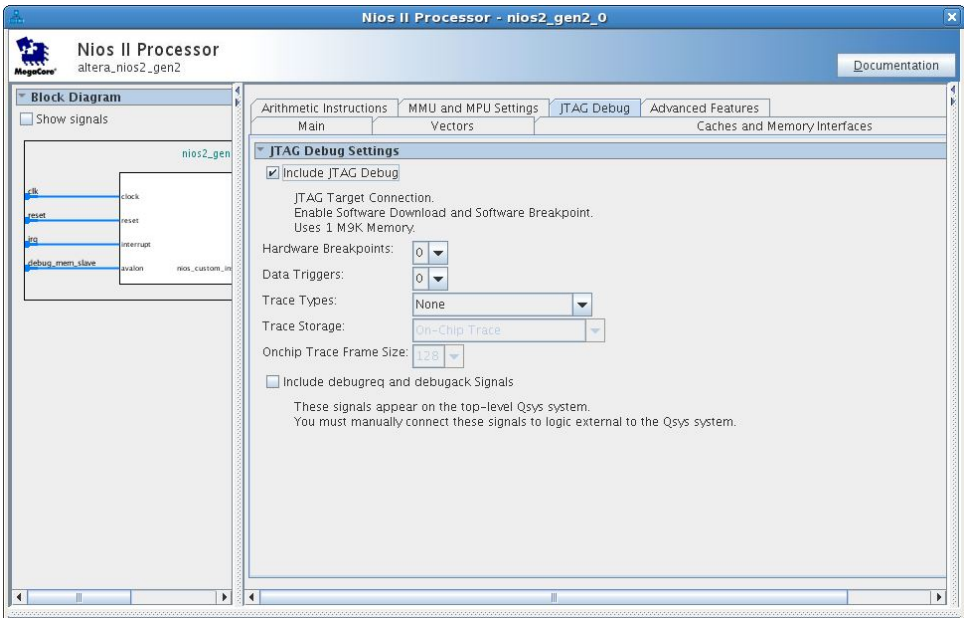
You can set hardware breakpoints on code located in read-only memory such as flash memory. If you set a breakpoint in a read-only area of memory, a hardware breakpoint, rather than a software breakpoint, is selected automatically.

To enable hardware breakpoints, you must configure the Nios II processor's debug level in Platform Designer to debug level 2 or higher. To configure the Nios II processor's debug level in Platform Designer to the appropriate level, perform the following steps:

1. On the Platform Designer **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 229. Nios II Processor — JTAG Debug Module — Platform Designer 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

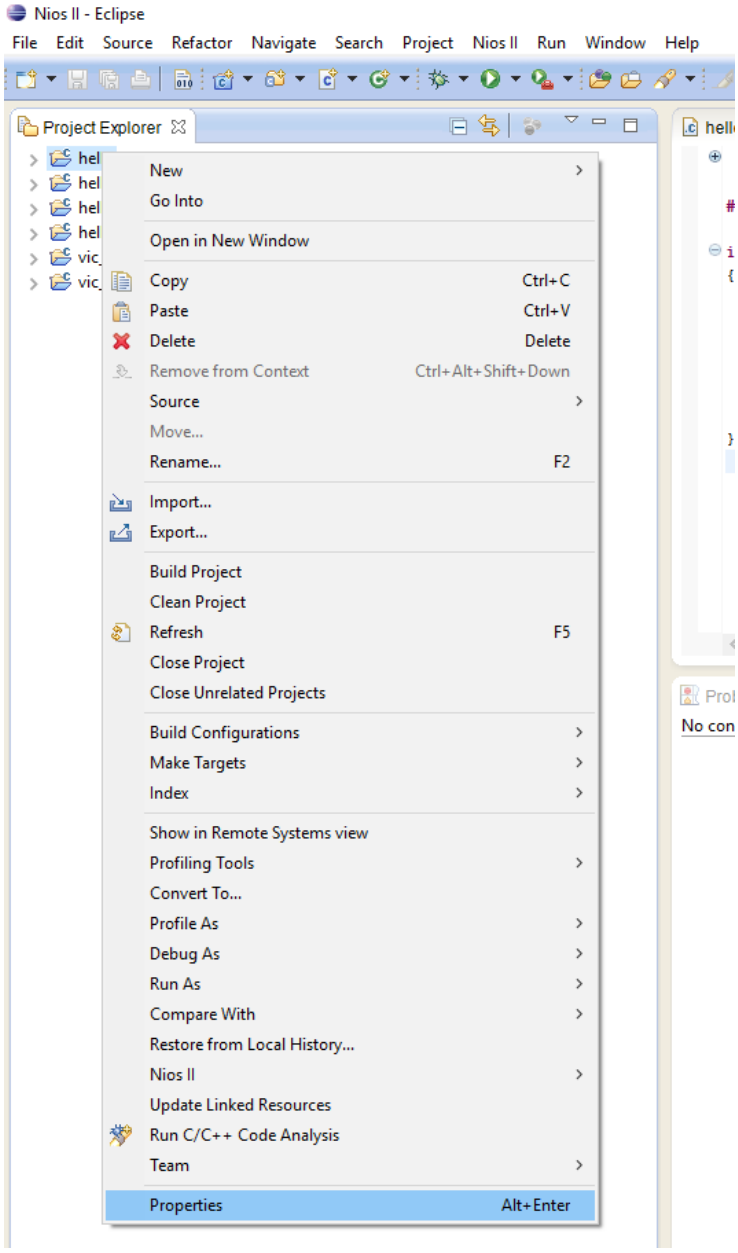
6.2.2.6. Debugger Stepping and Using No Optimizations

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:

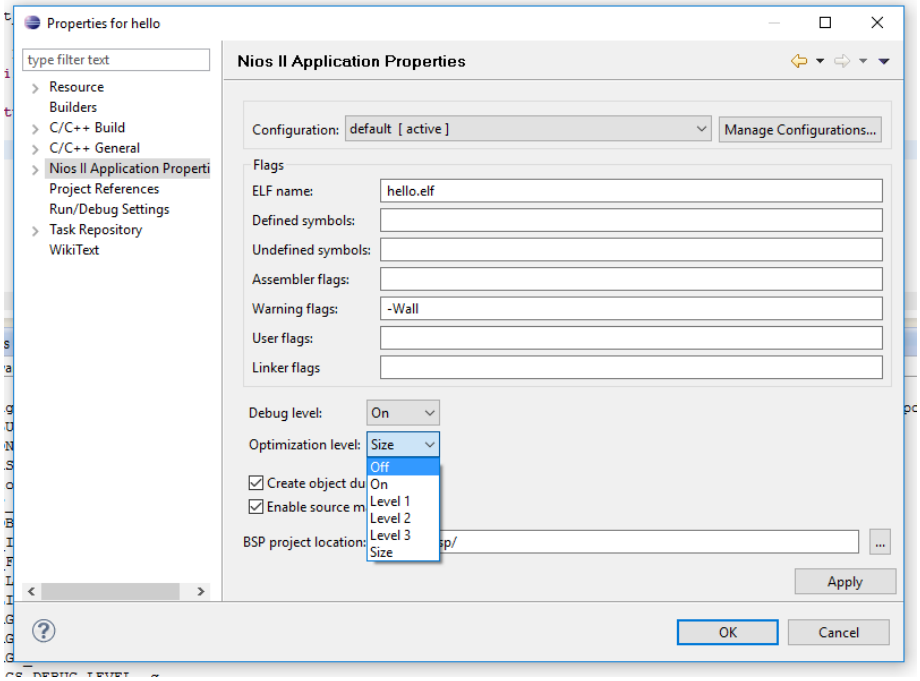
1. In the Nios II perspective, right-click your application project. A list of options appears.

Figure 230. 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 231. Nios Application Properties



4. In the **Optimization Level** list, select **Off**.
5. Click **Apply**.
6. Click **OK**

6.2.3. Using the Debug Code from Intel MAX 10 On-Chip Flash - User Flash Memory (UFM)

This section provides the steps for debugging a Nios II system where the code resides in on-chip flash. The debugging steps use the Nios II Software Build Tools. The selected boot option for the Nios II system is Execute-in-place.

Hardware requirements:

- Intel MAX 10 Development Kit (10M50DAF484C6GES)
- Power Adapter
- Intel FPGA Download Cable

Software requirements:

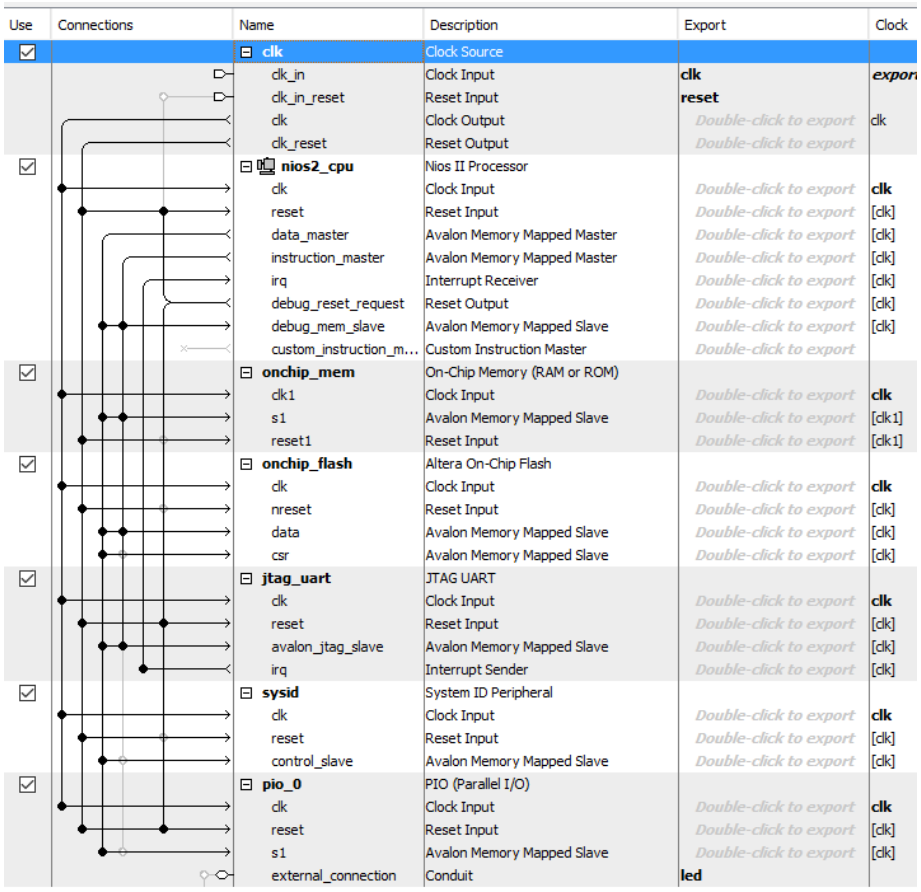
- Intel Quartus Prime Standard Edition, version 17.0
- Nios II Software Build Tools, version 17.0

6.2.3.1. Building Hardware Design in Platform Designer (Standard)

You must create a new Intel Quartus Prime project before including the hardware design components in the Platform Designer (Standard).

1. Open **Platform Designer (Standard)** from **Tool** menu in Intel Quartus Prime Standard Edition.
2. Add the following components to your system:
   - Nios II Processor IP core
   - On-Chip Memory (RAM or ROM) core
   - Intel FPGA On-Chip Flash IP core
   - JTAG UART core
   - System ID Peripheral core
   - PIO (Parallel I/O) core
3. Connect the components as shown below:

Figure 232. System Connections



Note: The `debug_reset_request` signal from Nios II processor must be connected to Nios II reset signal.

Before you generate the Platform Designer (Standard) system, you may have some warning and error messages. This is due to the overlap of base address, reset vector and exception vector in Nios II processor which are not set to any memory devices.

4. Double click on **Nios II Processor** IP component to edit its parameters.

5. Under the **Vector** tab:
   a. Select `onchip_flash.data` for **Reset Vector** to set the reset vector to On-chip Flash IP core.
   b. Select `onchip_flash.data` for **Exception Vector** to set the exception vector to On-chip Flash IP core.
6. Under the **JTAG Debug** tab:
   a. Select **Include JTAG Debug** and set the number of hardware breakpoint to 4.
7. Under the **System** tab, click on **Assign Base Addresses**.
8. Following the steps 4-7 clears all warning and error messages. Click **File**>**Save** to save changes.
9. Click **Generate HDL...** to generate the Platform Designer (Standard) system.

6.2.3.2. Compiling using the Intel Quartus Prime

1. Click on **Project**>**Add/Remove Files** to include the Platform Designer (Standard) generated `.qip` file. Set this `.qip` file as the top level file in your project.
2. Click on **Processing**>**Start**>**Start Analysis and Synthesis**.
3. Click on **Assignments**>**Pin Planner** to assign the pin location based on the Intel MAX 10 schematic for this design:
   - `reset` to `PIN_D9`
   - `clk` to `PIN_M9`
   - `LED[3]` to `PIN_AA21`
   - `LED[2]` to `PIN_U21`
   - `LED[1]` to `PIN_U22`
   - `LED[0]` to `PIN_T20`

   Close the pin planner.
4. Perform full compilation for your design.

6.2.3.3. Building Software Design

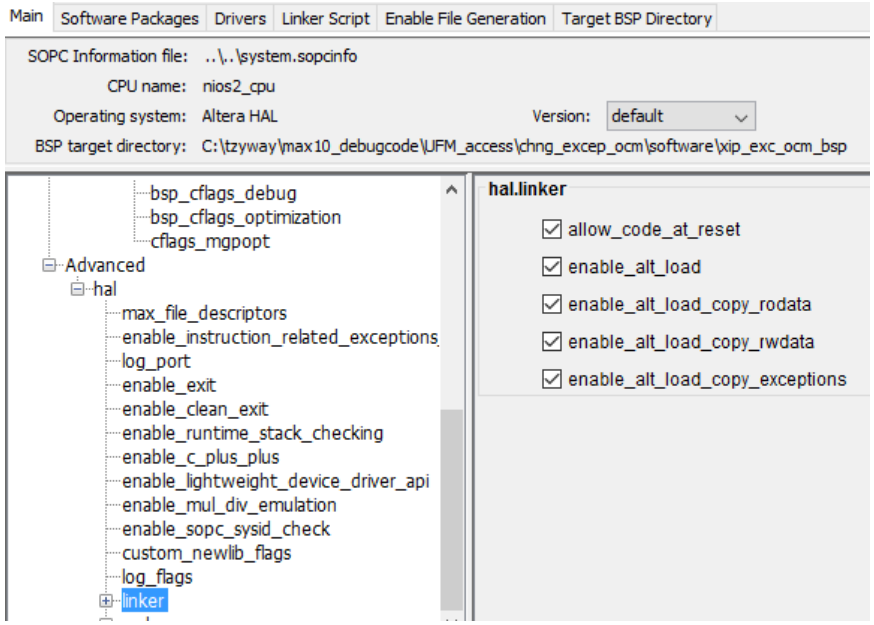
1. Open the **Nios II Software Build Tools for Eclipse** from **Tools** menu.
2. Create a workspace for your software project.
3. Click on **File**>**New**>**Nios II Application and BSP from Template** to create the application and BSP project. You can also choose to use the existing available template in your design.
4. Entering the **SOPC Information File Name** automatically detects the processor which is available in your design.
5. Enter a name for your project and click **Finish**. This step creates your application project with the linked BSP.
6. Select the **Hello World** template from the project template.
7. Modify the `hello_world.c` to instruct the Nios II processor to print sentences via `JtagUart` and write data to PIO for blinking the LED.

For example:

```
#include <stdio.h>
#include "system.h"
#include <io.h>
#include <unistd.h>
#include "altera_avalon_pio_regs.h"

int main()
{
  printf("Hello from Debug Test!\n");

  //blink the LED for two times
  for(int i=0;i<2;i++)
  {
      IOWR_ALTERA_AVALON_PIO_DATA(PIO_0_BASE, 0x1);
      usleep(1000000);
      IOWR_ALTERA_AVALON_PIO_DATA(PIO_0_BASE, 0x2);
      usleep(1000000);
      IOWR_ALTERA_AVALON_PIO_DATA(PIO_0_BASE, 0x4);
      usleep(1000000);
      IOWR_ALTERA_AVALON_PIO_DATA(PIO_0_BASE, 0x8);
      usleep(1000000);
  }

  printf("Does all the breakpoint is detected by debugger?\n");
  printf("Program ended!\n");

  return 0;
}
```

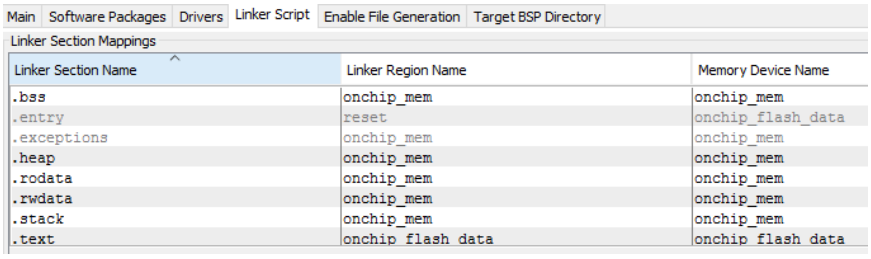
8. Right click on the application project, then click **Nios II**>**BSP editor**.
   a. Under the **Main** tab of the editor, click on **linker** and match the settings as shown below:

Figure 233. Linker Settings in Main Tab



   b. Under the **Linker Script** tab, change the `.text` section to `onchip_flash_data` as shown below:

Figure 234. Linker Section Mappings

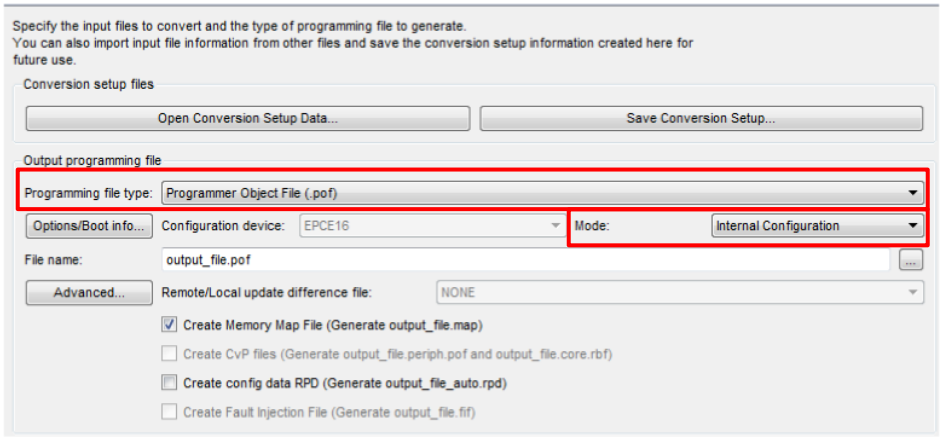


9. Right click on the application project, then click **Build Project**. After a successful built, the `.elf` file is created in the application folder.
10. Right click on the application project, then click **Make Target**>**Build**>**mem_init_generate** to create the `.hex` file.

6.2.3.4. Creating the POF File

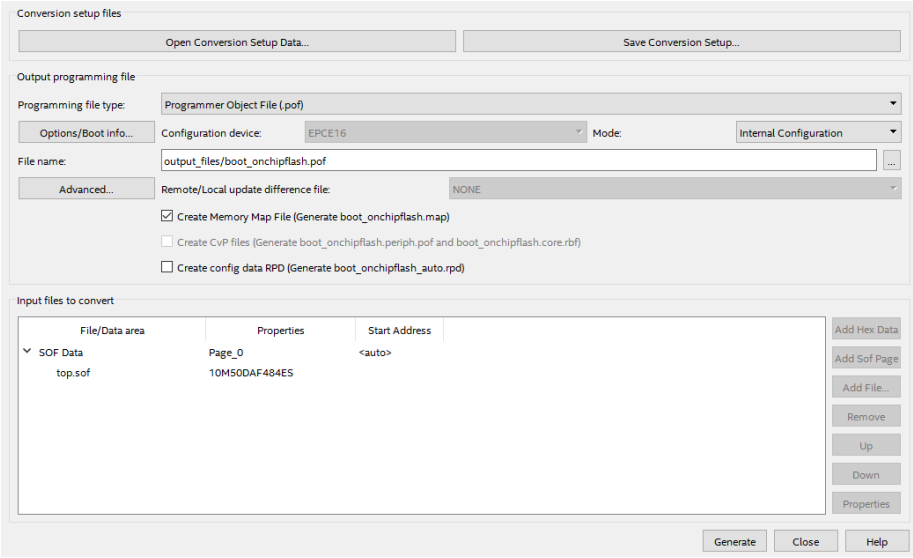
1. In Intel Quartus Prime, click on **Convert Programming Files** from the **File** tab.
2. Choose **Programmer Object File (.pof)** as **Programming file type:**.
3. Set **Mode** to **Internal Configuration**.

Figure 235. Convert Programming File Settings



4. Click on **Options/Boot info...**>**Load memory file for UFM source** and browse to the generated `.hex` file.
5. In the Convert Programming File dialog box, at the **Input files to convert** section, click **Add File...** and point to the generated Intel Quartus Prime .sof file.

Figure 236. Input Files to Convert in Convert Programming Files



6. Click **Generate** to create the .pof file.

6.2.3.5. Programming the POF File

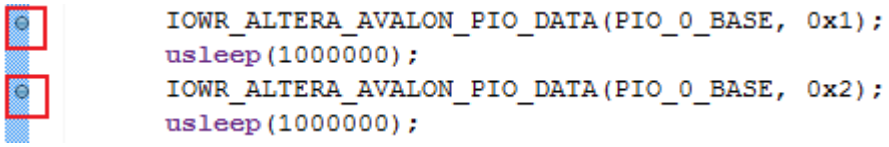
1. In Intel Quartus Prime, click on **Tools**>**Programmer**. Click on **Auto Detect** to detect the JTAG chain and device.
2. Add the generated `.pof` file in programmer and select the **Program/Configure** option.
3. Click **Start** to program the `.pof` file into the on-chip flash.

6.2.3.6. Debugging the Nios II Software Running in On-Chip Flash

After programming the `.pof` file into the On-chip flash, power cycle the board to ensure successful Nios II system boot up.

1. Open the **Nios II Software Build Tools for Eclipse** from **Tools** menu.
2. To set a breakpoint, double click the space beside each c programming code as shown below.

Figure 237. Step 2 Illustration

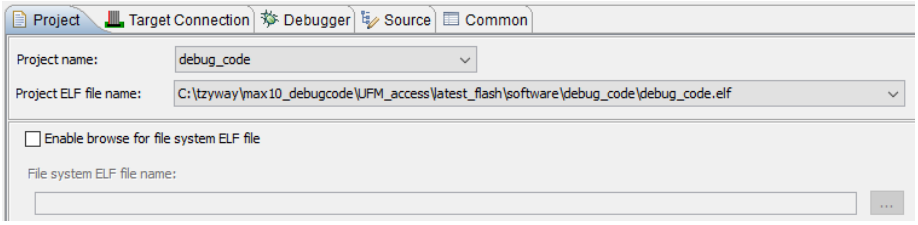


The build tool selects either hardware breakpoint or software breakpoint automatically based on the setting in BSP. Hardware breakpoint is used because the code is in On-chip flash. Only four hardware breakpoints can be used as based on parameter set in the Nios II IP.

3. Right click on the application project, then click on **Debug As**>**Debug Configuration**.

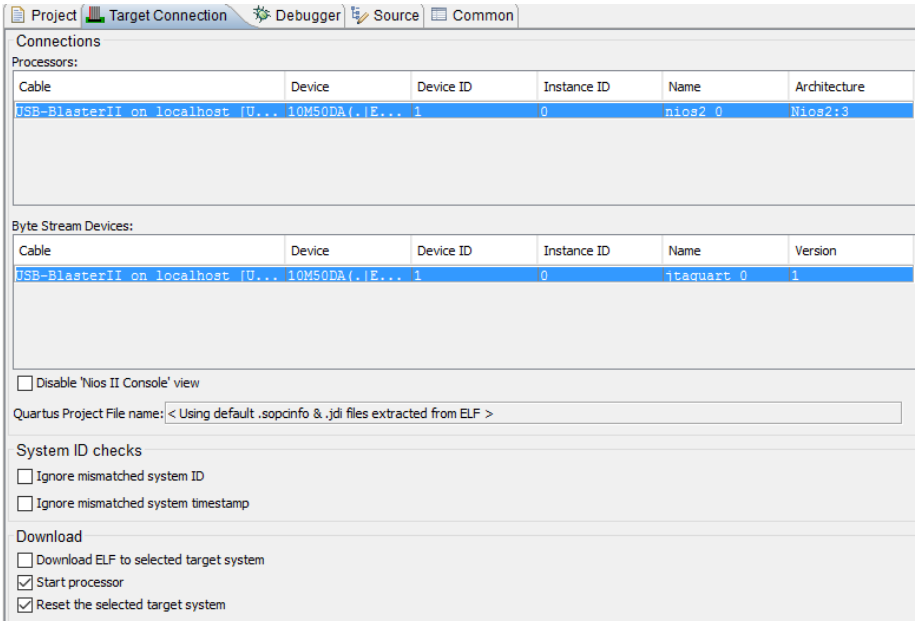
4. Under **Project** tab, select the correct project name if you have more than one project.

Figure 238. Step 4 Illustration



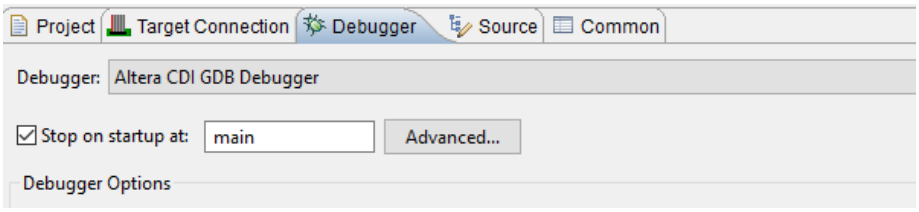
5. Under **Target Connection** tab, ensure that the Intel FPGA Download Cable II is detected for connection. Select **Start processor** and **Reset the selected target system** options. Un-select **Download ELF to selected target system**, as the Nios II processor is booting from on-chip flash, therefore downloading ELF is not required.

Figure 239. Step 5 Illustration



6. Under **Debugger** tab, you can choose to either stop the execution of the program at main or not.

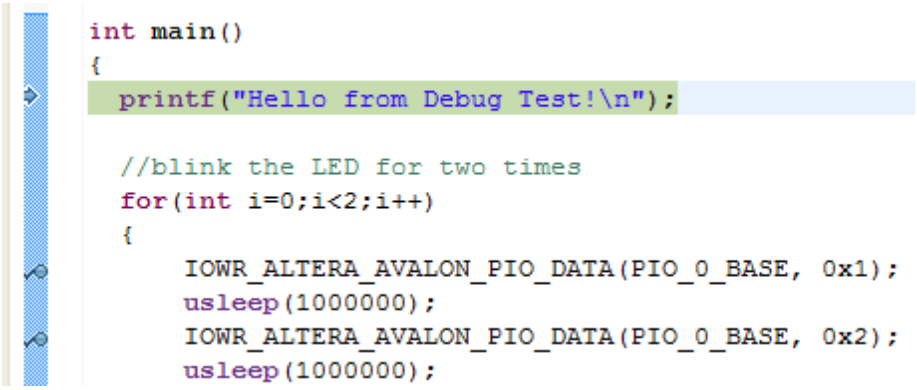
Figure 240. Step 6 Illustration



Note: If you are using one of the hardware breakpoint, this option is enabled.

7. Click on **Debug**. Click **Yes** when you are asked to switch to debug perspective. And the Nios II Debug window is opened.

Figure 241. Step 7 Illustration



The execution of the program is stopped at the main. The small arrow beside the code indicates the execution of the program stop at that line of program.

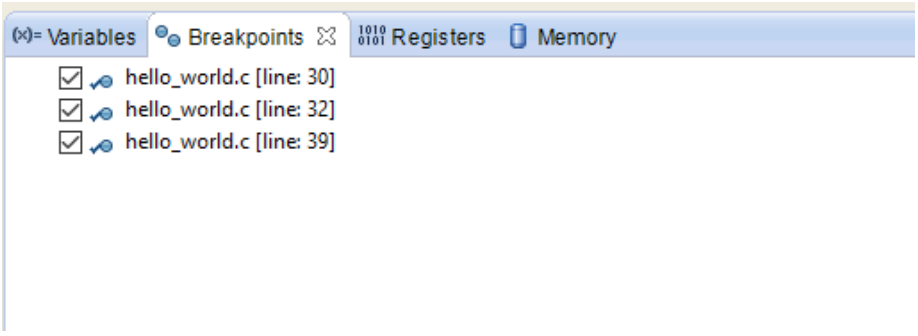
8. Use the different icons from the toolbar:
   - Resume icon: to continue the execution of the program. The execution of the program stops at the next breakpoint set in the program.
   - Step into or Step over icon: to debug the program line by line.
   - Terminate icon: to stop the debugging process

Figure 242. Step 8 Illustration



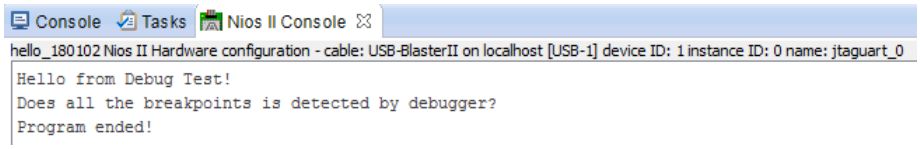
9. You can view the variables, breakpoints, register values and memory as shown below.

Figure 243. Step 9 Illustration



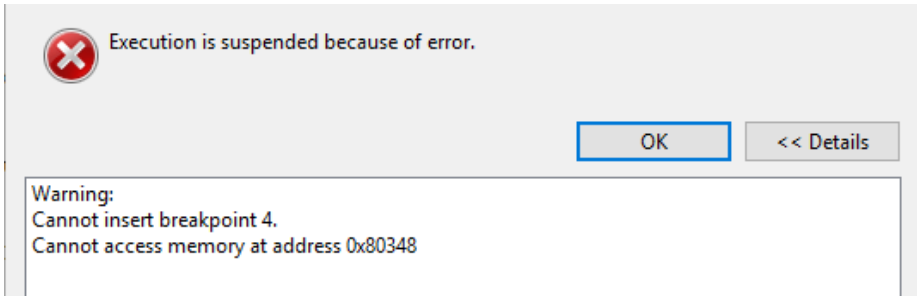
10. Nios II console displays all the messages from printf() function during program execution.

Figure 244. Step 10 Illustration



11. **Nios II Software Build Tools for Eclipse** reports an error if more than four set hardware breakpoints are inserted in the program.

Figure 245. Step 11 Illustration



6.3. Verification and Board Bring-Up

This section provides an overview of the tools available in the Intel 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

6.3.1. 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. Intel provides the following tools and strategies to help you overcome these difficulties:

- System Console
- Signal Tap II Embedded Logic Analyzer
- External Instrumentation
- Stimuli Generation

6.3.1.1. Prerequisites

To make effective use of this section, you should be familiar with the following topics:

- Defining and generating Nios II hardware systems with Platform Designer
- Compiling Nios II hardware systems with the Intel Quartus Prime development software

6.3.1.2. System Console

You can use the System Console to perform low–level debugging of a Platform Designer 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 Platform Designer components with interfaces that the System Console can use to send commands and receive data.

Table 45. Platform Designer 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 Platform Designer component provided by Intel, a custom component, or part of your Intel Quartus Prime project. However, this approach requires detailed knowledge of the JTAG commands.

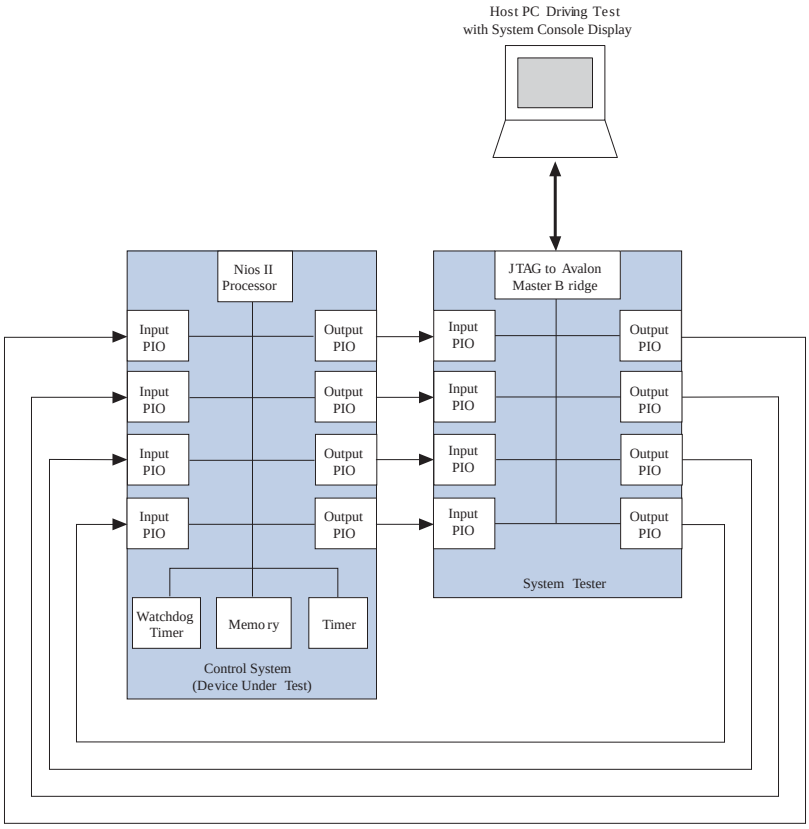
The System Console allows you to perform any of the following tasks:

- 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 246. Hardware-in-the-Loop Testing Using the System Console



To learn more about the System Console refer to the System Console chapter of the *Intel Quartus Prime Handbook Volume 3: Verification*.

Related Information

System Console

6.3.1.3. Signal Tap II Embedded Logic Analyzer

The Signal Tap II embedded logic analyzer is available in the Intel Quartus Prime software. It reuses the JTAG pins of the FPGA and has a low Intel 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 Signal Tap 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 Signal Tap 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 Intel Quartus Prime design software.

Because the Signal Tap 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 Signal Tap II embedded logic analyzer and co-verification, refer to *AN446: Debugging Nios II Systems with the Signal Tap II Embedded Logic Analyzer* and the *Intel Quartus Prime Handbook Volume 3: Verification*.

Related Information

- AN446: Debugging Nios II Systems with the Signal Tap II Embedded Logic Analyzer
- Intel Quartus Prime Handbook Volume 3: Verification

6.3.1.4. 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

Intel provides procedures to connect external verification devices such as oscilloscopes, logic analyzers, and protocol analyzers to your FPGA.

6.3.1.4.1. Signal ProbeSignal Probe

The Signal Probe 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 Signal Probe 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.

Intel recommends Signal Probe 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 Signal Probe
- Use the Signal Tap 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 Signal Tap II embedded logic analyzer is a viable solution if you do not require continuous sampling at a high rate. The Signal Tap 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 Signal Probe, refer to the Quick Design Debugging Using Signal Probe chapter in the *Intel Quartus Prime Handbook Volume 3: Verification*.

6.3.1.4.2. Logic Analyzer Interface

The Intel 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 Signal Tap 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 Intel Quartus Prime Logic Analyzer Interface, refer to the In-System Debugging Using External Logic Analyzers chapter in the *Intel Quartus Prime Handbook Volume 3: Verification*.

Related Information

In-System Debugging Using External Logic Analyzers

6.3.1.5. 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.

6.3.2. 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.

6.3.2.1. 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 395

6.3.2.1.1. 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 46. 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

6.3.2.1.2. 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 47. 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
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

6.3.2.1.3. 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

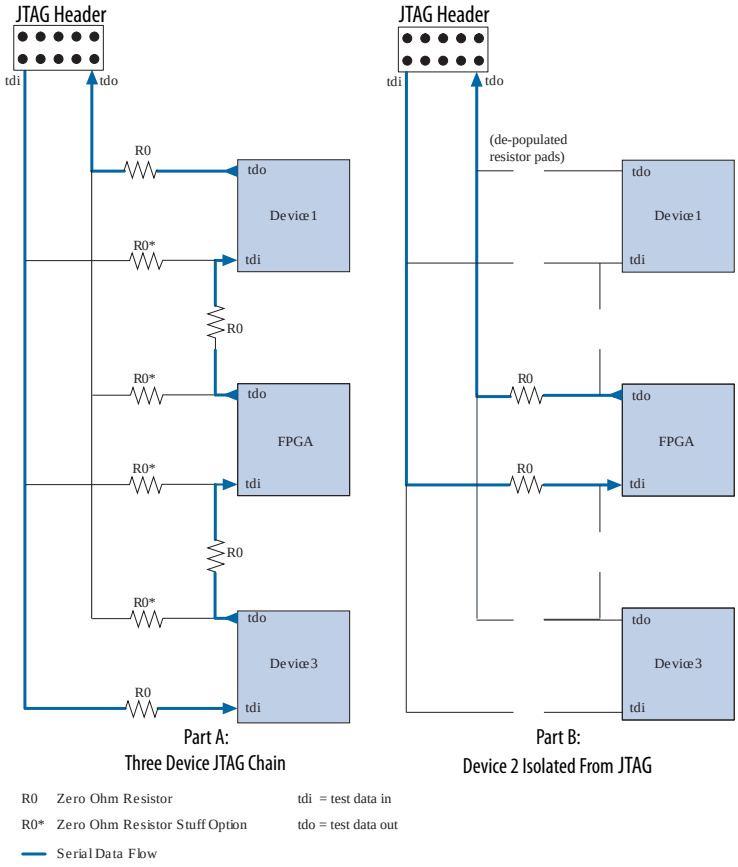
power supplies on your board. You can isolate your power supply by using fuses that are removed so that a stable external power supply can be substituted temporarily in your design.

6.3.2.1.4. 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 247. JTAG Isolation



To learn more about JTAG refer to *AN39: IEEE 1149.1(JTAG) Boundary-Scan Testing in Intel FPGA Devices*.

6.3.2.2. 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.

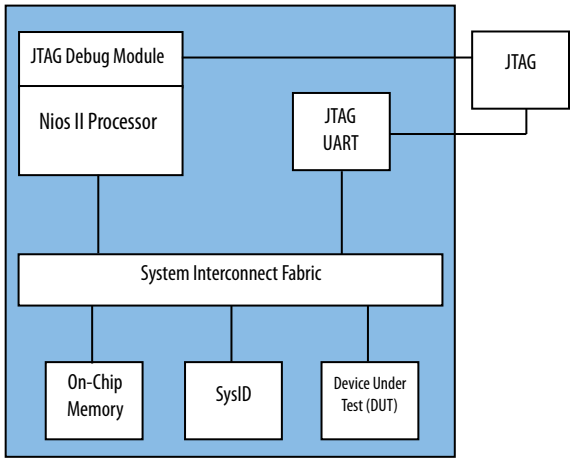
6.3.2.3. Minimal Test System

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 Intel Quartus Prime software.

Figure 248. Simple Test System



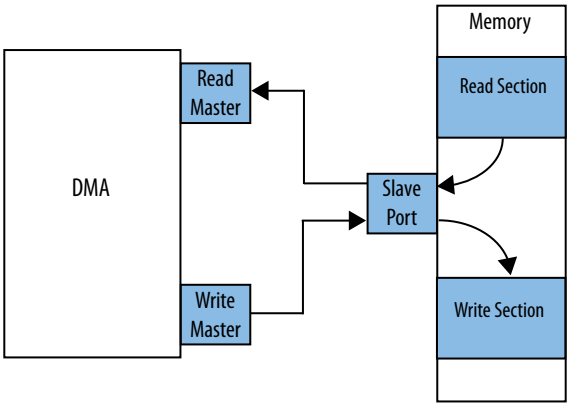
You can use the Nios II JTAG debug module to download software to the processor. Before testing any additional interfaces you should execute a small program that prints a message to the terminal to verify that your minimal system is functioning properly.

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 Signal Tap II embedded logic analyzer
- An external logic interface
- Signal Probe
- 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. Intel 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 249. 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 Gen2 Hardware Development Tutorial
- Quartus Prime Software: Verification

6.3.3. 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

6.3.3.1. 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.

6.3.3.2. Accelerating Verification

Intel 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.

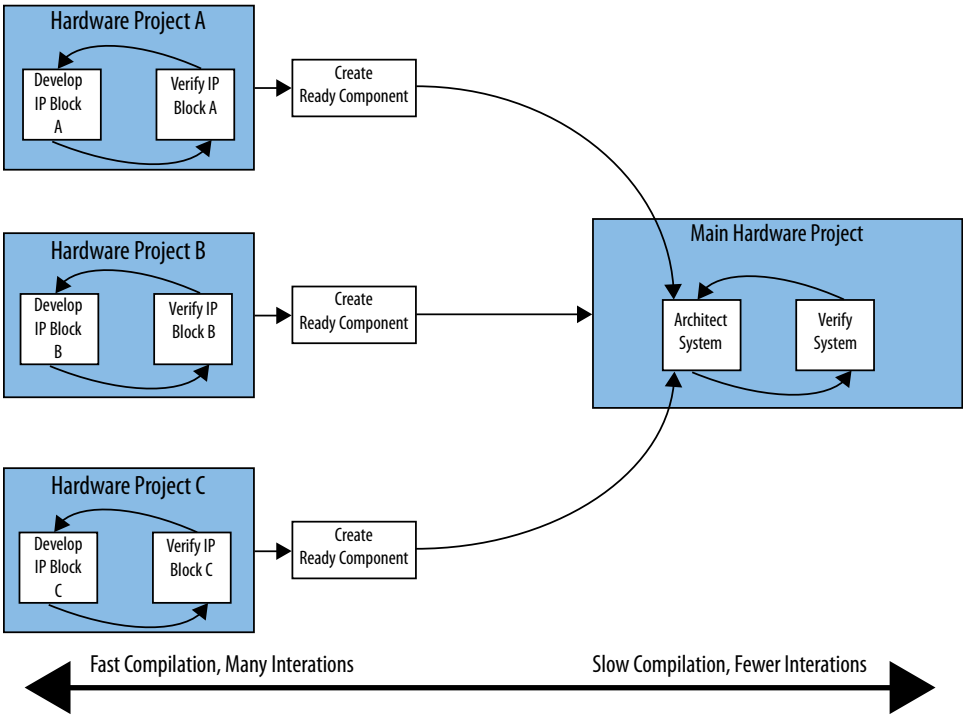
After the individual components are verified, you can integrate them in a Platform Designer system. The integrated system must include an Avalon-MM or Avalon Streaming (Avalon-ST) port. Using the component editor available from Platform Designer, you add an Avalon-MM interface to your existing component and integrate it in your system.

After your system is created in Platform Designer, 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 250. IP Verification and Integration Flow



To learn more about component editor and system integration, refer to the following documentation:

Related Information

- Creating Platform Designer Components
- Avalon Interface Specifications

6.3.3.3. Using Software to Verify Hardware

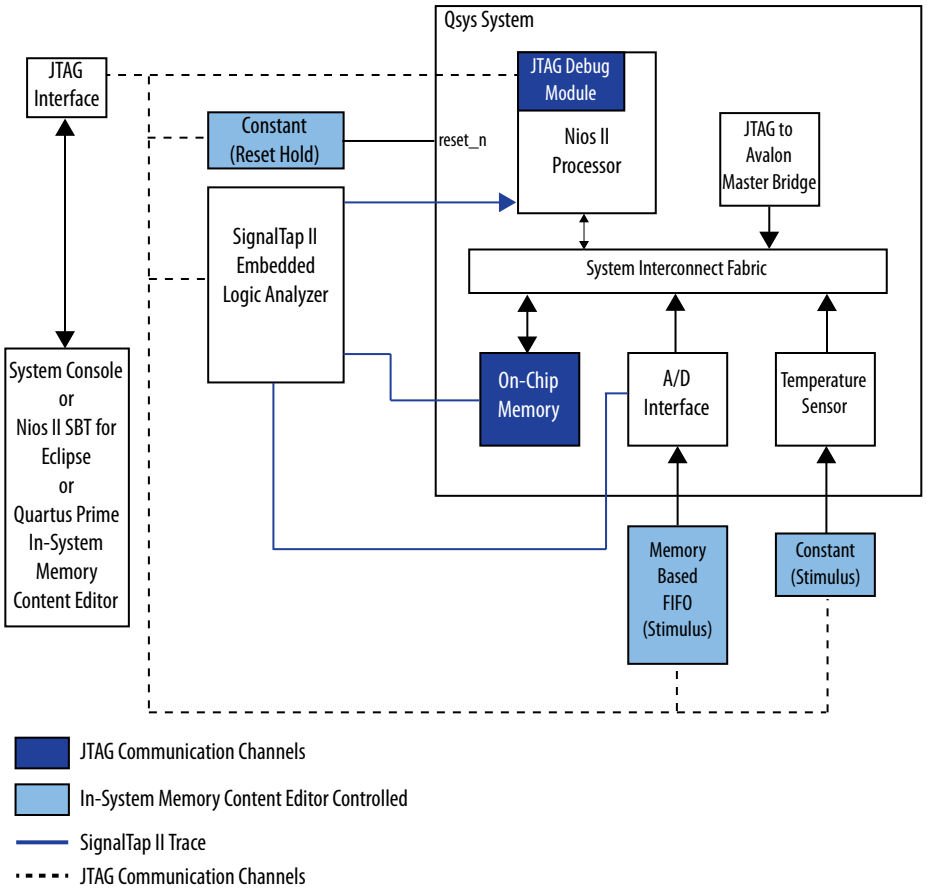
Many hardware developers use test benches and test harnesses to verify their logic in simulations. These strategies can be very time consuming. Instead of relying on simulations for all your verification tasks, you can test your logic using software or scripts, as the figure below illustrates.

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 Intel 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 Intel 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 251. 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

6.3.3.4. 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.

6.4. Additional Embedded Design Considerations

Consider the following topics as you design your system:

- JTAG signal integrity
- Extra memory space for prototyping
- System verification

6.4.1. 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, Intel 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 Intel Quartus Prime Handbook.

6.4.2. Memory Space For System Prototyping

Even if your final product includes no off-chip memory, Intel 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.

6.5. Simulating Nios II Embedded Processor Designs

This section describes the process of generating an RTL simulation environment with Nios II example designs, Platform Designer, and the Nios II Software Build Tools (SBT) for Eclipse. This application note also describes the process of running the Nios II RTL simulation in the ModelSim Edition simulator.

The increasing pressure to deliver robust products to market in a timely manner has amplified the importance of comprehensively verifying embedded processor designs. Therefore, consider the verification solution supplied with the processor when choosing an embedded processor. Nios II embedded processor designs support a broad range of verification solutions, including the following:

- **Board Level Verification**—Intel offers a number of development boards that provide a versatile platform for verifying both the hardware and software of a Nios II embedded processor system. You can use the Nios II SBT for Eclipse with its built-in debugger to verify designs running on either development or custom boards. You can further debug the hardware components that interact with the processor with the Signal Tap II embedded logic analyzer.
- **Register Transfer Level (RTL) Simulation**—RTL simulation is a powerful means of debugging the interaction between a processor and its peripheral set. When debugging a target board, it is often difficult to view signals buried deep in the system. RTL simulation alleviates this problem as it enables you to functionally probe every register and signal in the design. You can easily simulate Nios II-based systems in the ModelSim simulator with an automatically generated simulation environment that Platform Designer and the Nios II SBT for Eclipse create.

Related Information

- Nios II Classic Software Developer’s Handbook
- Nios II Gen2 Software Developer's Handbook

6.5.1. Before You Begin

This document assumes that you have prior experience using Platform Designer as well as a familiarity with the ModelSim simulator. In order to simulate the Nios II design using the instructions in this document, you must have the following software installed:

- Intel Quartus Prime software
- ModelSim
- Nios II Embedded Design Suite

6.5.2. Setting Up and Generating Your Simulation Environment in Platform Designer

To open the example design, perform the following steps:

1. Download the **an351_design.zip** design example from the Simulating Nios II Embedded Processor Design page on the Intel website, and then extract the design example to your hard drive. The location to which you extract the file is referred to as <your project directory> throughout the remainder of this document.
2. Start the Intel Quartus Prime software.
3. On the File menu, click Open Project.
4. Browse to *<your project directory>*/**an351_design**.
5. Select **an351_project.qpf**.
6. Click **Open**.
7. On the Tools menu, click **Platform Designer**.
8. Open the **niosii_system.qsys** file.

   Note: The design example used for this application note is a complete Platform Designer system. Ensure that you have completed building your Platform Designer system before you start to generate the simulation models.
9. On the **Generation** tab, set the following parameters to these values:
   - Create simulation model—None
   - Create testbench Platform Designer system—Simple, BFMs for clocks and resets

     Note: If your system has exported ports other than the clock and reset, choose Standard, BFMs for standard Avalon interfaces.
   - Create testbench simulation model—Verilog
   - Create HDL design files for synthesis—Turn off
   - Create block symbol file (**.bsf**)—Turn off
10. Click **Generate**. Save the system if prompted.

Related Information

Simulating Nios II Embedded Processor Designs

6.5.2.1. Platform Designer-Generated System Simulation Files

At this point in the design flow, Platform Designer has generated your system and created all of the files necessary for simulation listed in the table below. These simulation files are located in the *<your project directory>*/**an351_design/niosii_system/testbench** directory.

Table 48. Platform Designer Files Generated for Nios II Simulation

File	Description
Platform Designer testbench system files	Platform Designer generates a testbench system when you enable the Create testbench Platform Designer system option. Platform Designer connects the corresponding

continued...

File	Description
	Avalon Bus Functional Models to all exported interfaces of your system. For more information about Platform Designer, refer to the System Design with Platform Designer section in volume 1 of the Intel Quartus Prime Handbook.
msim_setup.tcl	Sets up a ModelSim simulation environment and creates alias commands to compile the required device libraries and system design files in the correct order, and loads the top-level design for simulation.
Memory Initialization Files (**.mif**)	Creates Memory Initialization Files (**.mif**) to initialize memory components in your system. Use Nios II SBT for Eclipse to create Nios II processor program to populate the **.mif** files.

6.5.2.2. Memory Simulation Models

You can use two types of memory models for simulation purposes: generic and vendor-specific. For Intel-provided memory controllers, you are provided with generic simulation models. If you are using custom memory controllers, you should use the simulation models the memory controller vendor provides. This application note discusses the generic memory model.

When you simulate a Nios II design that involves external memory interfaces using the flow with Nios II SBT for eclipse, an abstract DRAM memory model will be inserted at the Avalon-MM slave interface of the UniPHY IP.

This model is added in front of Avalon MM interface of a memory controller to act as the memory interface with a pre-initialize memory content. Essentially, this model operates as a bridge between the memory interface and the Avalon-MM interface of the memory controller.

For write operation, the data is written to both internal abstract DRAM memory model and external memory. For read operation, the data is replaced by the read data from abstract DRAM memory model.

6.5.2.3. Using IP and Platform Designer Simulation Setup Scripts

Intel IP cores and Platform Designer systems generate simulation setup scripts. Modify these scripts to set up supported simulators.

6.5.3. Creating the Nios II Software

This section describes how to finish setting up your simulation by using the Nios II SBT for Eclipse or Nios II command shell to create a software test project and to generate the necessary files for initializing the memories used in your simulation.

- For creating the project using Nios II SBT tool, refer to the Creating a Nios II SBT Eclipse Project.
- For creating the project using command line, refer to the Creating a software project with command line.

6.5.3.1. Creating a Software Project Using Nios II SBT for Eclipse

In this application note, you simulate a simple Hello World program with Platform Designer. The Hello World software prints a message to the console via JTAG UART. To create and build the software project, perform the following steps:

1. Open the Nios II SBT for Eclipse version 11.0 or later.
2. On the **File** menu, point to **New**, and click **Nios II Application and BSP from Template**.
3. Select the SOPC Information (**.sopcinfo**) file name by browsing to *<your project directory>*/**an351_design**, and then select **niosii_system.sopcinfo**.
4. For **Project Name**, type **hello_world_an351**.
5. Select **Hello World** from the **Templates** option.
6. Click **Finish**.
7. Right-click on **hello_world_an351** in Project Explorer and then click **Build Project**.

   Now you have successfully built the Hello World project. In the next step, you will invoke ModelSim simulation from the Nios II SBT for Eclipse. This function populates the Memory Initialization File (**.mif**) with the Hello World program and starts the ModelSim software.
8. Right-click on **hello_world_an351** in Project Explorer. Point to **Run As**, and then click **Nios II ModelSim**.

6.5.3.2. Creating a Software Project Using Command Line

In this section, you create a simple **Hello World** project using command line. To create and build the `hello_world` example for modelsim simulation, perform the following steps:

1. Open the Nios II Command Shell.
2. Change to the directory where you store your design generated by Platform Designer. The following steps refer to this directory as `<projects>`.
3. The `<projects>` directory contains a subdirectories named **software**. Change the directory to this folder by using the command `cd software`.
4. Create a folder named **bsp** in this folder by using the command `mkdir bsp`.
5. Change your directory into this **bsp** folder using the command `cd bsp` and create the bsp using the command `nios2-bsp hal . ../../<sopcinfo>`.
6. Change your directory into the app folder using the command `cd ../app` and create the app make file by using the command `nios2-app-generate-makefile --bsp-dir ../bsp --app-dir . --src-dir .`
7. Compile the source code by using the command `make all`.
8. Generate the memory file (.hex) by using the command `make mem_init_generate`.
9. Now, you have successfully built the project. The next step is invoke the modelsim simulation using the command `vsim &`.

6.5.4. Running Simulation in the ModelSim Simulator Using Nios II SBT for Eclipse

After you have launched the ModelSim simulator from the Nios II SBT for Eclipse, ModelSim automatically compiles the required device libraries and system design files, and elaborates and loads the top-level design. The msim_setup.tcl script creates alias commands for each of the steps. These commands are listed in the table below.

Table 49. Nios II Alias Commands

Macros	Description
`dev_com`	Compiles device library files.
`com`	Compiles the design files in correct order.
`elab`	Elaborates the top-level design.
`elab_debug`	Elaborates the top-level design with the **novopt** option.
`ld`	Compiles all the design files and elaborates the top-level design.
`ld_debug`	Compiles all the design files and elaborates the top-level design with the **novopt** option.

Run the simulation in the ModelSim simulator by performing the following steps:

1. In the ModelSim software, on the **File** menu, click **Load**. Browse to *<your project directory>*/**an351_design** and select **wave.do**. This step opens a waveform viewer with all the JTAG UART signals.
2. In the Transcript window, type `run 2 ms`. This step starts the simulation for two milliseconds.

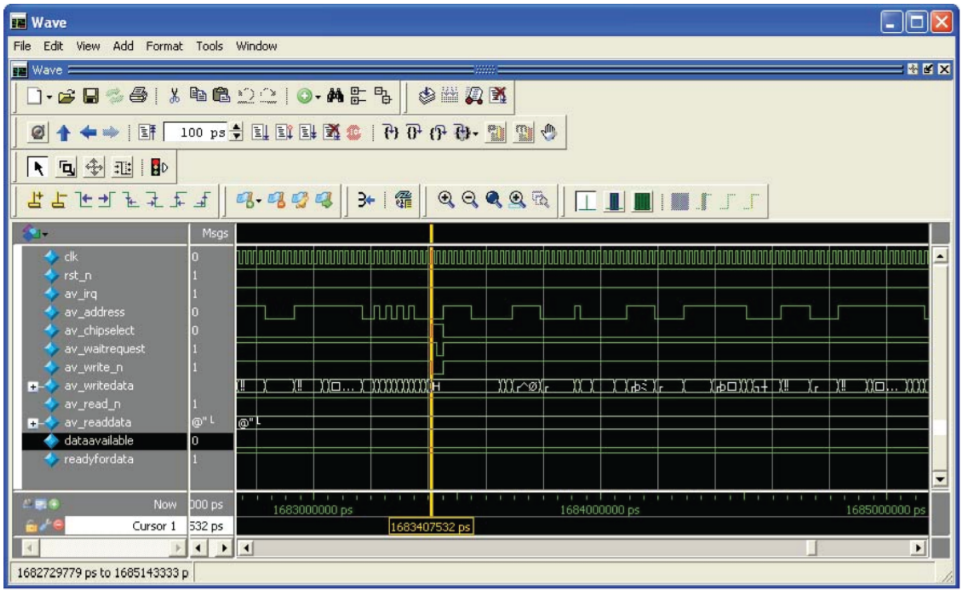
6.5.5. Running Simulation in the ModelSim Simulator Using Command Line

After you have launched the Modelsim simulator using command `vsim &` in the Nios II Command Shell, you can run the simulation in the modelsim simulator by performing the following steps:

1. In the transcript window, change your working directory to mentor by using the command `cd <qsys_name>/testbench/mentor`.
2. Now, modify the `msim_setup.tcl` by including the command below into the section copy ROM/RAM files to simulation directory `file copy -force<projects>/software/app/mem_init/<qsysname_ram.hex> ./`
3. Run the msim_setup.tcl by using the command `do msim_setup.tcl`.
4. Compiles all the design files and elaborates the top-level design with novopt option by using command `ld_debug`.
5. Load the wave.do file provided by using the command `do <projects>/wave.do`.
6. Type `run 2ms` to start the simulation for 2 milliseconds

At the end of the simulation, you should see a "Hello from Nios II!" message in the Transcript window. You can observe the simulation results from the waveform viewer as well. The figure below shows the simulation result. The waveform is zoomed in at a specific simulation time in which the Nios II processor writes the first H character to the JTAG UART component.

Figure 252. Simulation Results



6.6. Nios II Debug, Verification, and Simulation Revision History

Document Version	Changes
2018.11.12	Added section: *Using the Debug Code from Intel MAX 10 On-Chip Flash - User Flash Memory (UFM)*.
2017.11.06	Added new topics: • *Creating a Software Project Using Command Line*. • *Running Simulation in the ModelSim Simulator Using Command Line*.
2017.06.12	Added section: *Simulating Embedded Processor Design.*
2016.12.19	Initial release.

Related Information

Document Revision History for Embedded Design Handbook on page 6

7. Optimizing Nios II Based Systems and Software

Optimizing techniques enable better performance and resource utilization in your design. This chapter provides various optimizing techniques from the hardware and software perspective.

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

7.1. Hardware Acceleration and Coprocessing

This section discusses how you can use hardware accelerators and coprocessing to create more efficient, higher throughput designs in Platform Designer. 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

7.1.1. 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.

7.1.1.1. 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. Platform Designer 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:(27)
    - 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 Intel FPGAs that have DSP Blocks.
    - Embedded Multipliers—Includes dedicated embedded multipliers available on the target device. This option is available only on Intel 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. Platform Designer facilitates experimenting with the balance of software and hardware implementation. You can even design custom hardware accelerators for specific system tasks.

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 Platform Designer Components" chapter in the *Intel Quartus Prime Handbook Volume 1: Design and Synthesis*.

Related Information

- Creating Platform Designer Components
- Nios II Custom Instruction User Guide
- Hardware Acceleration and Coprocessing on page 442

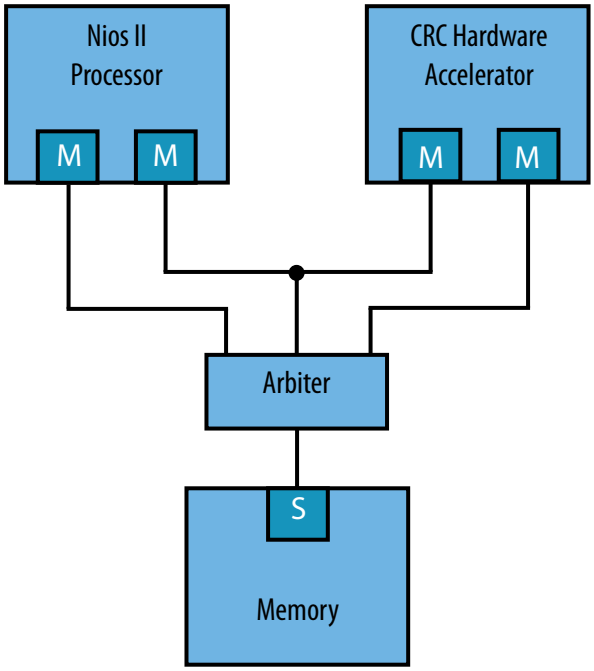
7.1.1.2. Accelerating Cyclic Redundancy Checking (CRC)

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 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.

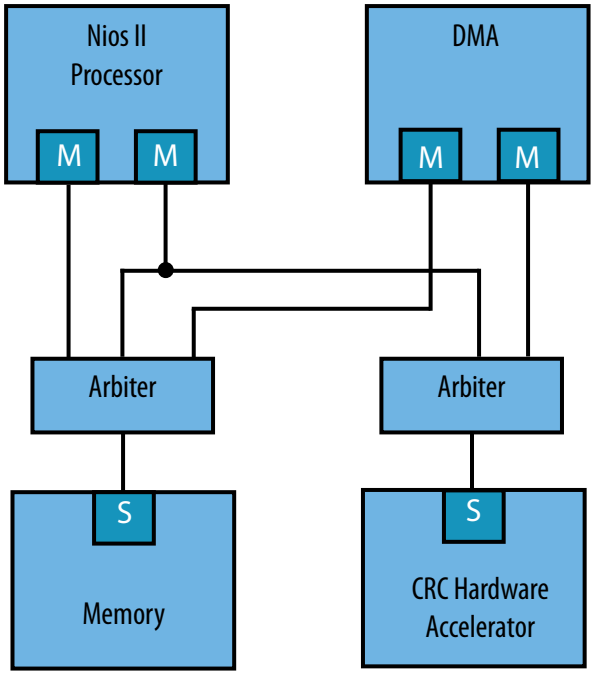
(27) The Nios II/s core is only available with Nios II Classic.

Figure 253. 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 254. 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 Platform Designer system using the component editor.

To learn more about using component editor, refer to the "Component Editor" section in the Creating Platform Designer Components chapter of the *Quartus Prime Handbook Volume 1: Design and Synthesis*. You can find additional examples of hardware acceleration on the Intel IP: Reference Designs web page.

Related Information

- Intel FPGA IP: Reference Designs
- Creating Platform Designer Components

7.1.1.2.1. 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.

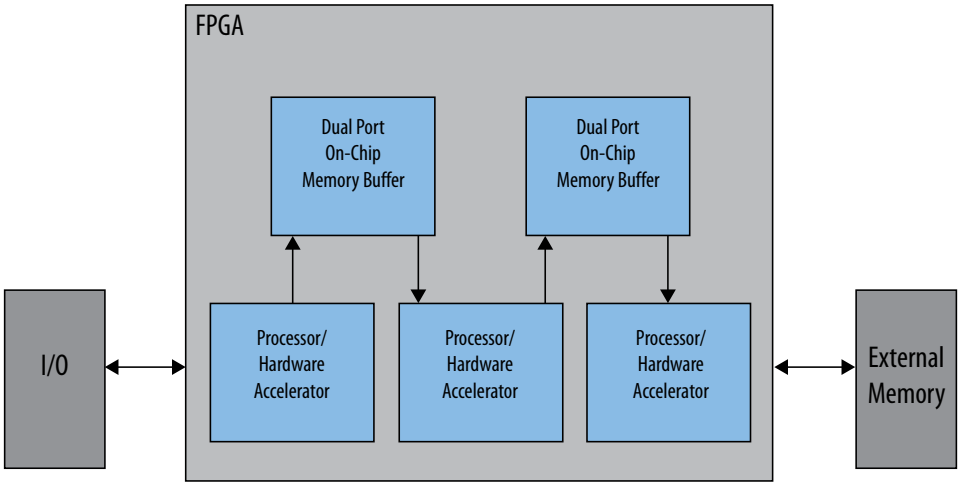
Typically, memory interfaces cause the most problems in systems that contain multiple processors and hardware accelerators. The following recommendations for interface design can maximize the throughput of your hardware accelerator:

- 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.

7.1.1.2.2. 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 255. 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

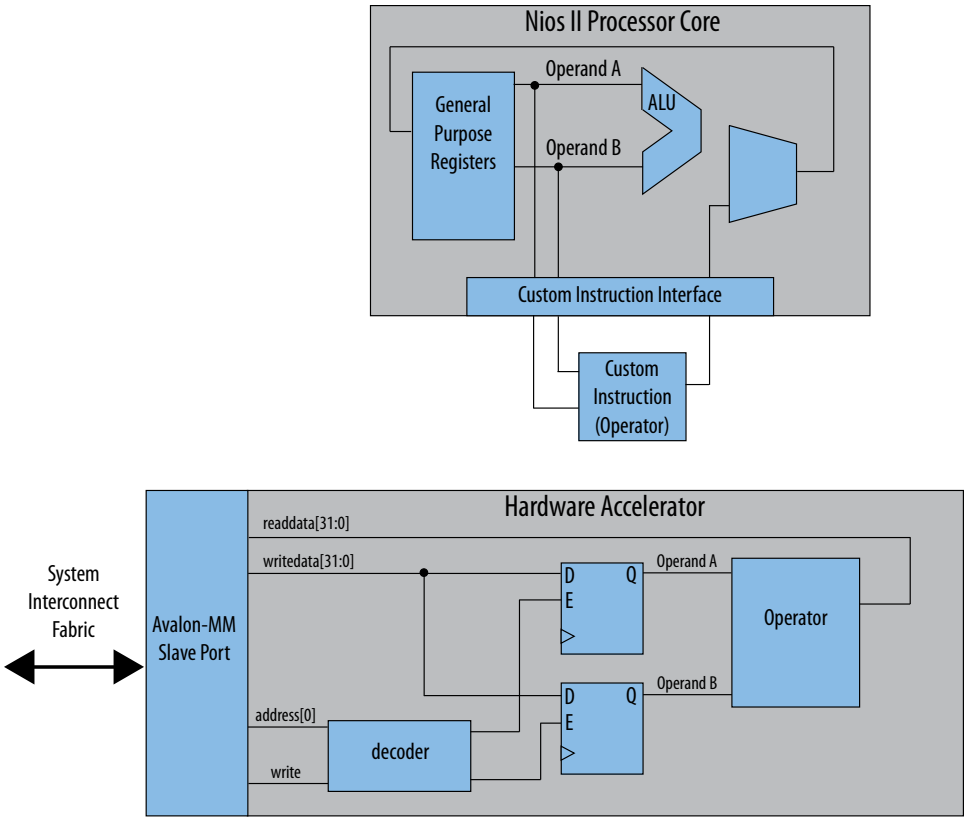
Memory System Design on page 52

7.1.1.3. 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).

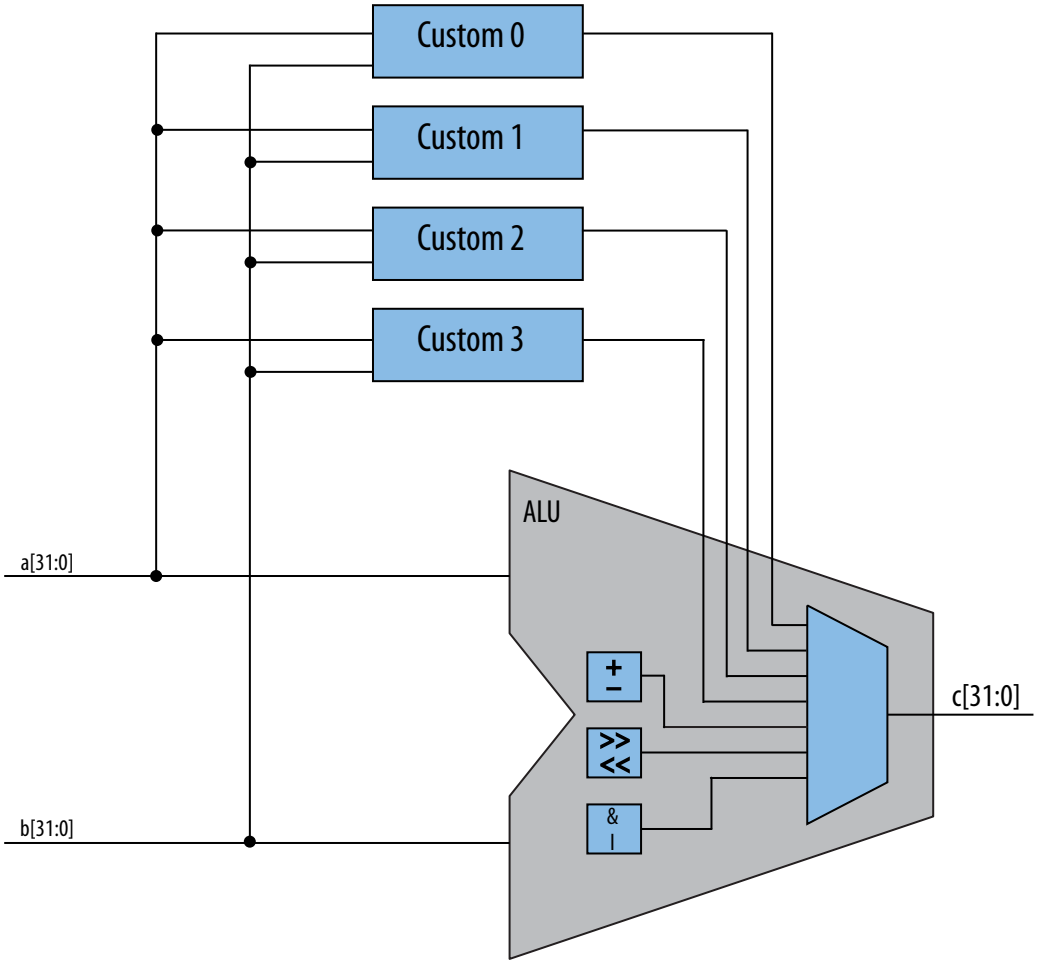
All custom instructions have a similar structure. They include up to two data inputs and one data output, and optional clock, reset, mode, address, and status signals for more complex multicycle operations. If you need to add hardware acceleration that requires many inputs and outputs, a custom hardware 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 256. Implementation Differences between a Custom Instruction and Hardware Accelerator



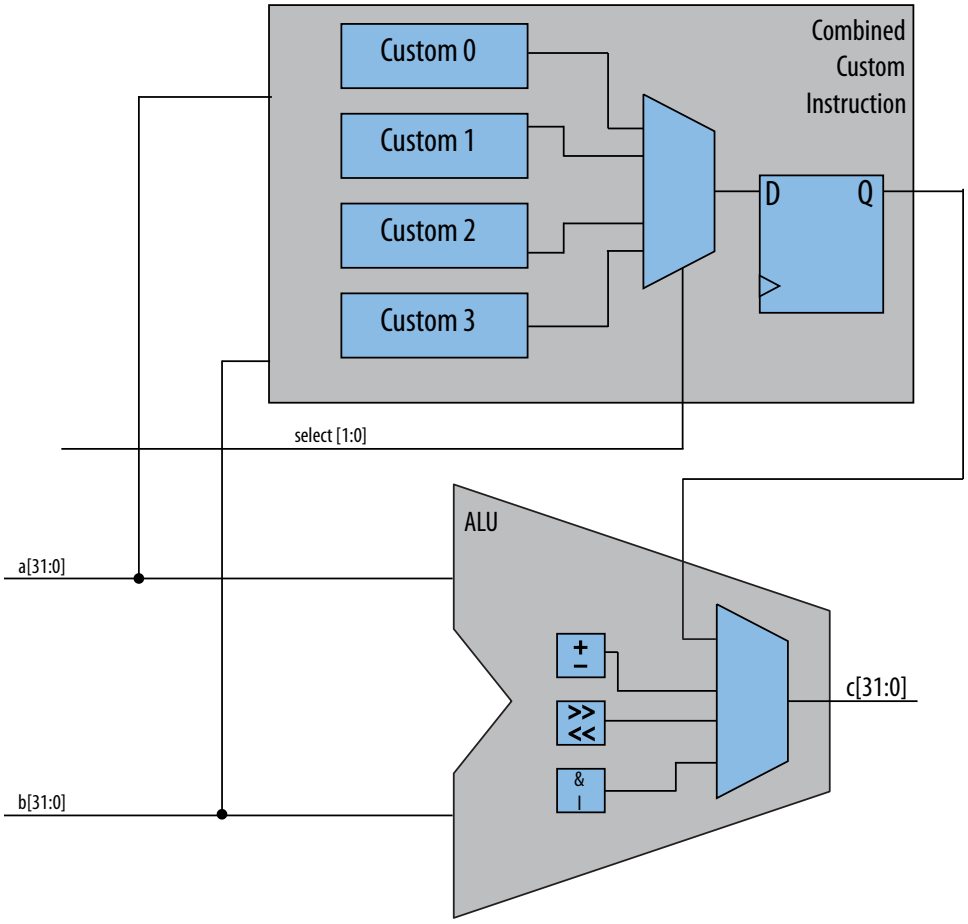
Because custom instructions extend the Nios II processor's ALU, the logic must meet timing or the fMAX of the processor will suffer. As you add custom instructions to the processor, the ALU multiplexer grows in width as the figure below illustrates. This multiplexer selects the output from the ALU hardware (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 257. 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.

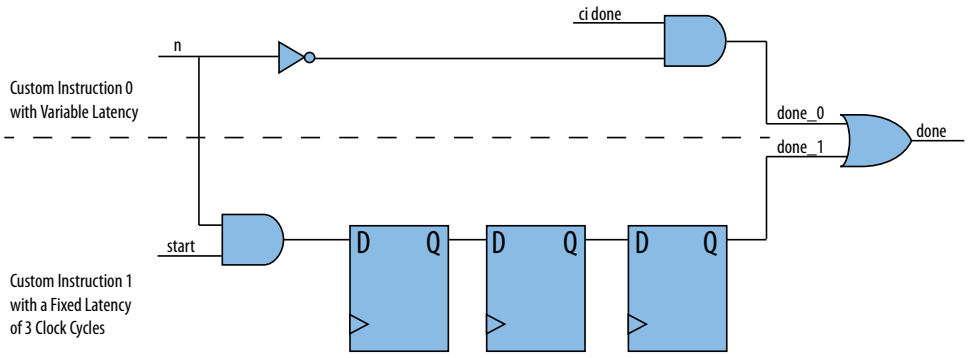
Figure 258. Combined Custom Instruction



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 259. 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

7.1.2. 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.

7.1.2.1. 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.

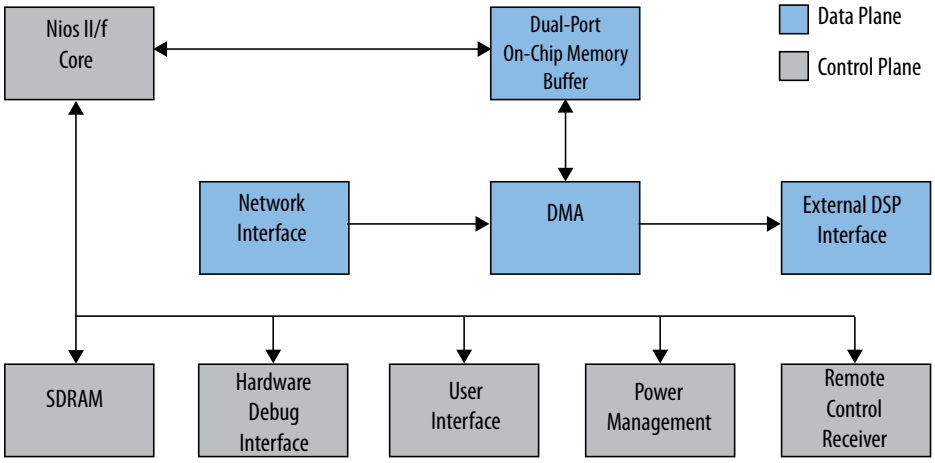
When a design includes more than one processor you must partition the algorithm carefully to make efficient use of all of the processors. The following example includes a Nios II-based system that performs video over IP, using a network interface to supply data to a discrete DSP processor. The original 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 260. Over-Utilized Video System

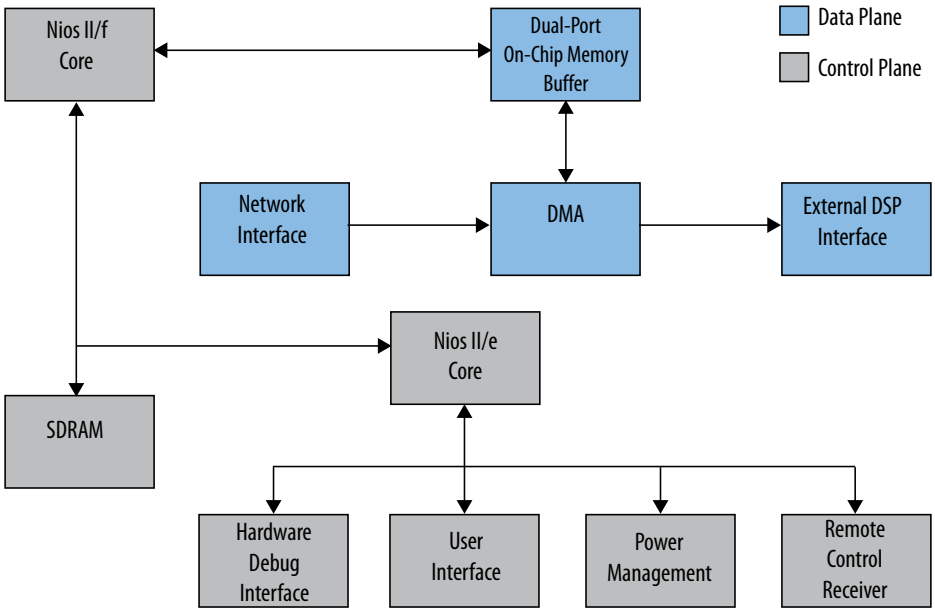


Adding a second Nios II processor to the system, allows the workload to be divided so that one processor handles the network functionality and the other the control interfaces.

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 Intel website.

Figure 261. 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 have uninterrupted Nios II/f core, set memory arbiter to give low priority to the Nios II/e 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

7.1.2.1.1. Accessing Multiple Nios II Designs

You use the `device` and `instance` arguments from the command line to select the desired Nios II core.

Program the FPGA device with a `.sof` file that contains a design with two Nios II instances:

```
[]$ quartus_pgm -m jtag -c 1 -o "p;output_files/top.sof"
```

Use system console to decode the Nios II instance to SLD node:

```
[]$ system-console --cli

% design_load ./output_files/top.sof
```

```
% get_service_paths processor
{/devices/1SX280LH(2|3)|1SX280LN2(|AS)|..@1#1-4#Stratix 10L SoC Dev Kit/(link)/
JTAG/alt_sld_fab_0_alt_sld_fab_0_sldfabric.node_0/nios2_gen2_0.data_master} {/
devices/1SX280LH(2|3)|1SX280LN2(|AS)|..@1#1-4#Stratix 10L SoC Dev Kit/(link)/
JTAG/alt_sld_fab_0_alt_sld_fab_0_sldfabric.node_1/nios2_gen2_1.data_master}
```

Here, you can see that the two Nios II cores enumerate to these SLD nodes:

```
nios2_gen2_0 = node_0
nios2_gen2_1 = node_1
```

Also, the two Nios II cores on JTAG chain with `jtagconfig`:

```
[]$ jtagconfig -n
1) Stratix 10L SoC Dev Kit [1-4]
  C321D0DD   1SX280LH(2|3)/1SX280LN2(|AS)/..
    Design hash    BE8F2B069D867486EF17
    + Node 19104600  Nios II #0
    + Node 19104601  Nios II #1
    + Node 0C006E00  JTAG UART #0
    + Node 0C006E01  JTAG UART #1
```

Interact with Nios II instance 0:

```
[]$ nios2-download -c 1 -d 1 -i 0 -s 0x81030 -I 0xdecadeca -t 0x0 --stop
Using cable "Stratix 10L SoC Dev Kit [1-4]", device 1, instance 0x00
Pausing target processor: OK
Reading System ID at address 0x00081030: verified
Leaving target processor paused
```

Interact with Nios II instance 1:

```
[]$ nios2-download -c 1 -d 1 -i 1 -s 0x10 -I 0xcafec0de -t 0x0 --stop
Using cable "Stratix 10L SoC Dev Kit [1-4]", device 1, instance 0x01
Pausing target processor: OK
Reading System ID at address 0x00000010: verified
Leaving target processor paused
```

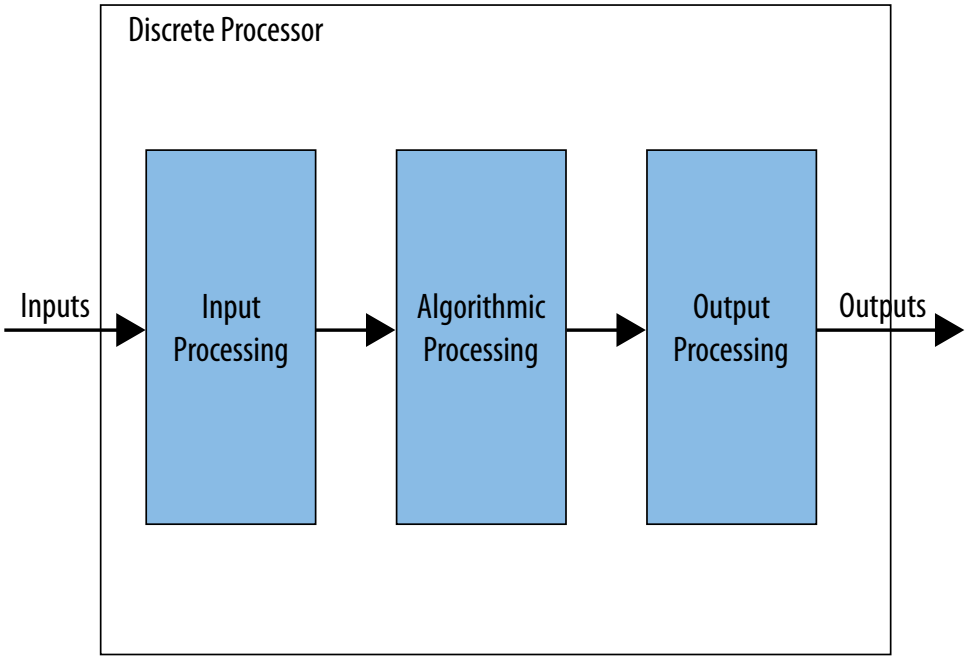
7.1.2.2. Pre- and Post-Processing

The high performance video system illustrated in Figure 261 on page 453 distributes the workload by separating the control and data planes in the hardware. Figure 263 on page 456 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

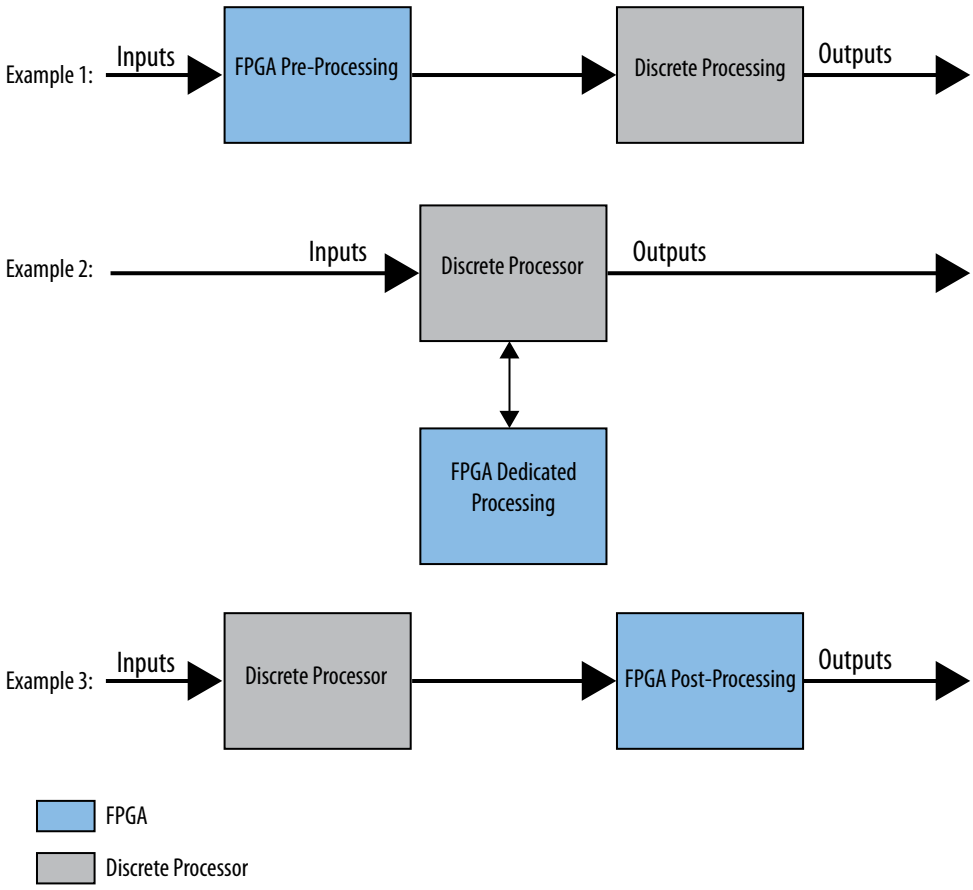
By off loading the processing required for the inputs or outputs to an FPGA, the discrete processor has more computation bandwidth available for the algorithmic processing.

Figure 262. 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 263. Pre- Dedicated, and Post-Processing



7.1.2.3. 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.

7.1.2.3.1. Low-Speed State Machines

Low-speed state machines are typically used to control peripherals. The Nios II/e processor pictured in Figure 260 on page 452 could implement a low speed state machine to control the peripherals.

Even though the Nios II/e core does not include a data cache, Intel 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 Platform Designer 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 Platform Designer are no different than those you would use if you were creating a state machine manually. One noticeable difference in the Platform Designer 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

7.1.2.3.2. 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

7.1.2.3.3. 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.

7.2. Software Application Optimization

This section examines techniques to increase your software application's performance and decrease its size.

7.2.1. 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.

7.2.2. 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.

7.2.2.1. Analyzing the Problem

The first step to accelerate your system processing is to identify the slowest task in your system. Intel 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

7.2.2.2. 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

7.2.2.2.1. 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 Platform Designer. 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 Platform Designer:
  - **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

- Using Tightly Coupled Memory with the Nios II Processor Tutorial on page 483
- Nios II Software Build Tools Reference

- Instantiating the Nios II Gen2 Processor
- Cache and Tightly-Coupled Memory

7.2.2.2.2. 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 442
- Nios II Custom Instruction User Guide

7.2.2.2.3. Improving Data Movement

If your application performs many sequential data movement operations, a DMA peripheral might increase the efficiency of these operations. Intel 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) Controller Core—Descriptor-based DMA peripheral that can perform multiple operations before being serviced by processor. Intel recommends to use the later Modular Scatter-Gather DMA (mSGDMA) Core for your new designs.

  For more information, refer to the Intel FPGA Scatter-Gather DMA chapter in the *Embedded Peripherals IP User Guide*.
- Modular Scatter-Gather DMA (mSGDMA) Core—Modular descriptor-based DMA peripherals that facilitates easy integration with the FPGA fabric, offload processor by performing data movement operations with preloaded instructions. The core consists of a dispatcher block with optional read master and write master blocks. The descriptor block receives and decodes the descriptor, and dispatches instructions to the read master and write master blocks for further operation.

  For more information, refer to the Intel FPGA Modular Scatter-Gather DMA chapter in the *Embedded Peripherals IP User Guide*.

Related Information

- Embedded Peripherals IP User Guide
- HAL Peripheral Services on page 165

7.2.3. 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.

7.2.3.1. 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 Intel literature pages.You can download the design files from the Documentation: Tutorials page of the Intel 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 on page 483

- FPGA Documentation

7.2.3.2. 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

7.2.4. 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.

7.2.4.1. 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 33. 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
```

7.2.4.2. 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 50. 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.

7.3. Memory Optimization

This section presents tips and tricks that can be helpful when implementing any type of memory in your Platform Designer system. These techniques can help improve system performance and efficiency.

7.3.1. 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.

7.3.2. Match Master and Slave Data Width

Matching the data widths of master and slave pairs in Platform Designer is advantageous. Whenever a master port is connected to a slave of a different data width, Platform Designer 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.

7.3.3. 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 Platform Designer, 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 Platform Designer. 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.

7.3.4. 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.

7.3.5. 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.

7.4. 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.

7.4.1. 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 Intel website.

Download the design example file, and unzip the file into a working directory.

Related Information

Nios II Ethernet Acceleration Design Example

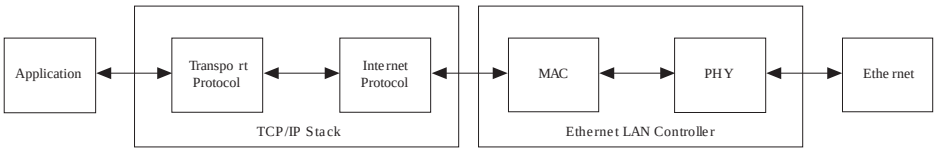
7.4.2. The Structure of Networking Applications

This section describes the different parts of a general networking application.

7.4.2.1. Ethernet System Hierarchy

The figure below shows the flow of information from an embedded networking application to the Ethernet.

Figure 264. 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.

7.4.2.2. 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.

7.4.2.3. 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

7.4.3. 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.

7.4.3.1. User Application Optimizations

This section describes some effective ways to decrease the amount of time your application uses the Nios II processor.

7.4.3.1.1. 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 Intel'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 474

7.4.3.1.2. 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 442

7.4.3.1.3. 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 51. 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.

7.4.4. 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 Intel 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 Intel FPGA Triple Speed Ethernet Intel FPGA IP 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**.

7.4.4.1. 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 470

7.4.4.2. NicheStack Specific Optimizations

This section describes the targeted optimizations that you can use to increase the performance of the NicheStack networking stack directly.

7.4.4.2.1. NicheStack Thread Priorities

Intel'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()`.

7.4.4.2.2. 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.

7.4.4.2.3. 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 Intel 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 Intel 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.

7.4.4.2.4. Super Loop Mode

Although the Intel-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, Intel does not support this configuration.

7.4.5. 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.

7.4.5.1. 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.

7.4.5.2. Network Interface (Intel FPGA Triple Speed Ethernet Intel FPGA IP Function)

The Nios II EDS supports the Intel FPGA Triple Speed Ethernet Intel FPGA IP function. The Triple Speed Ethernet Intel FPGA IP function's role is essentially to translate an application's Ethernet data into physical bits on the Ethernet link. The Triple Speed Ethernet Intel FPGA IP function supports 10/100/1000 Mb networks. The table below lists the key design parameters that impact network performance.

Table 52. Triple Speed Ethernet Intel FPGA IP Function

Parameter	Intel FPGA Triple Speed Ethernet Intel FPGA IP 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	Intel FPGA 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 Intel FPGA IP function is capable of sending and receiving Ethernet data quickly because of the Scatter-Gather DMA peripherals. The Triple Speed Ethernet Intel FPGA IP function also allows you to select from a flexible range of send and receive FIFO depths.

7.4.5.3. NicheStack Device Driver Model

The NicheStack networking stack presents a simplified device driver model for integrating Ethernet devices, and the Intel FPGA Triple Speed Ethernet Intel FPGA IP function solution is fully optimized to support this model.

In the Triple Speed Ethernet Intel FPGA IP 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 Intel FPGA IP 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 Intel FPGA IP function device driver results in an overall performance increase in the system.

For information about the Triple Speed Ethernet Intel FPGA IP function, refer to the Triple Speed Ethernet Intel FPGA IP 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

7.4.6. 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.

7.4.6.1. 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 Intel 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.

7.4.6.2. Test Setup

The benchmarking tests were conducted between a workstation and an Intel development board. The Intel 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.

7.4.6.2.1. 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.

7.4.6.3. Test Methodology

This section describes the parameters used in the benchmarking tests.

7.4.6.3.1. 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.

7.4.6.3.2. 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.

7.4.6.3.3. 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>);
```

7.4.6.3.4. 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.

7.4.6.4. 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.

7.4.6.4.1. 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

7.4.6.4.2. 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.

7.4.6.4.3. 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.

7.4.6.4.4. 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 `-03` 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`.

7.4.6.5. 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.

7.4.7. 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.

7.5. Using Tightly Coupled Memory with the Nios II Processor Tutorial

This document describes how to use tightly coupled memory in designs that include a Nios® II processor and discusses some possible applications. It also includes a tutorial that guides you through the process of building a Nios II system with tightly coupled memory.

The Nios II architecture includes tightly coupled master ports that provide guaranteed fixed low-latency access to on-chip memory for performance-critical applications. Tightly coupled master ports can be connected to instruction memory and data memory, to allow fixed low-latency read access to executable code as well as fixed low-latency read or write access to data. Tightly coupled masters are dedicated instruction or data master ports on the Nios II core, which is different from the embedded processor's instruction and data master ports.

This document assumes you are familiar with the Nios II tightly coupled memory. For more information, refer to the Processor Architecture chapter in the *Nios II Gen2 Processor Reference Handbook*.

Related Information

Using Tightly Coupled Memory with Nios II Processor - Design Example Webpage

7.5.1. Reasons for Using Tightly Coupled Memory

You can implement one or more of the following functions or modules using tightly coupled memory to enhance the performance of your system:

- Separate exception stack for use only while handling interrupts
- Fast data buffers
- Fast sections of code
  - Fast interrupt handler
  - Critical loop
- Constant access time that is guaranteed not to have arbitration delays

For programs with modest memory requirements, all of the code and data can be held in a tightly coupled memory pair.

7.5.2. Tradeoffs

There are design tradeoffs when using tightly coupled memory, including the following:

- You must balance the benefits of more general speed enhancement provided by a cache, averaged over time, with the specific dedicated memory block consumed by a tightly coupled memory whose sole purpose is to make a single part of the code faster.
- Software guarantees that performance-critical code or data is located in tightly coupled memory. That particular piece of code or data achieves high performance. Locating the code within tightly coupled memory eliminates cache overhead such as cache flushing, loading, or invalidating.
- You must divide on-chip memory equitably to provide the best overall combination of tightly coupled instruction memory, tightly coupled data memory, instruction cache, and data cache.

7.5.3. Guidelines for Using Tightly Coupled Memory

This section details the guidelines and limitations that should be taken into consideration when designing the hardware and software with tightly coupled memory.

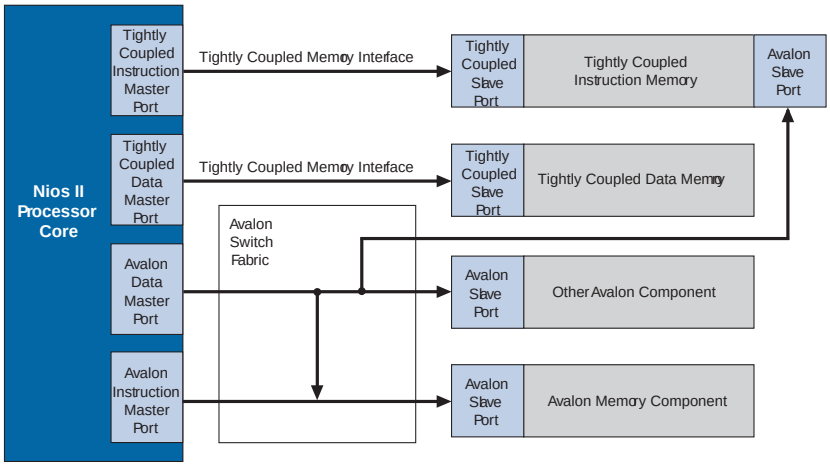
7.5.3.1. Hardware Guidelines

The following guidelines apply to Nios II hardware designs that include tightly coupled memory:

- Tightly coupled masters are presented as additional master ports on the Nios II processor.
- An On-Chip Memory component is the only memory that can connect to a tightly coupled master port on the Nios II core.
- A tightly coupled master on a processor must connect to exactly one on-chip memory slave port. This slave port cannot be shared by any other master port.
- Each on-chip memory can be connected to only one tightly coupled master even if it is a dual port memory.
- The availability of the data and instruction master ports for the tightly coupled memory is dependent on the type of Nios II core used.
- When using the On-Chip Memory component as a tightly coupled memory for Nios II, you must always create it as a RAM, not as a ROM. Tightly coupled memories configured as ROM would result in failure.
- To conserve the logic elements, use one 2-kilobyte (KB) tightly coupled memory, rather than two 1-KB tightly coupled memories.

The figure below is a block diagram of a simple Nios II system that includes tightly coupled memories and other Platform Designer System Integration Tool components.

Figure 265. Nios II System with Tightly Coupled Instruction and Data Memory



7.5.3.2. Software Guidelines

The following two guidelines apply to Nios II software that uses tightly coupled memory:

- Software accesses tightly coupled memory addresses just like any other addresses.
- Cache operations have no effect when targeting tightly coupled memory.

7.5.3.2.1. Locating Functions in Tightly Coupled Memory

Assigning data to a tightly coupled data memory also involves using a section attribute. Alternatively, you can include the tightly coupled memory as a `#define` in the **system.h** file. The name of the memory is followed by _BASE and is used as a pointer to reference the tightly coupled data memory.

The software example in this tutorial provides a source code example showing how to locate a particular source code function in a particular linker section. A function is declared to reside within a linker section with the C section attribute in the file **timer_interrupt_latency.h**. This C header file locates `timer_interrupt_latency_irq()` in the `.exceptions` section as follows:

```
externvoid timer_interrupt_latency_irq (void* base, alt_u32 id)__attribute__
((section
(".exceptions")));
```

The Nios II Software Build Tools (SBT) creates linker sections for each memory module in the system. A source code function can be located within a particular tightly coupled instruction memory simply by assigning that function to the linker section created for that tightly coupled instruction memory.

The Nios II SBT creates additional linker sections with address mappings that are controlled by Platform Designer. For the `.exceptions` section, the physical address offset and memory module in which to base that linker section is manipulated through Platform Designer. You locate the `.exceptions` section in a memory module covered by a tightly coupled data memory using the **Exception Vector** field found on the **Core Nios II** tab of the configuration wizard.

For more information about the C section attribute, refer to the Developing Programs Using the Hardware Abstraction Layer chapter in the *Nios II Software Developer's Handbook*.

7.5.4. Tightly Coupled Memory Interface

The term tightly coupled memory interface refers to an Avalon-like interface that connects one master to one slave. Refer to Figure 265 on page 485. Tightly coupled memory interfaces connect tightly coupled masters to their tightly coupled slaves. Tightly coupled memory interfaces are designed to be connected to one port of an on-chip memory device. These devices are known as "altsyncrams" to Verilog HDL and VHDL designers.

7.5.4.1. Restrictions

You must observe the following restrictions when designing with tightly coupled memories:

- Tightly coupled slaves must be on-chip memories.
- Only one master and one slave can be connected to a given tightly coupled memory interface, which makes the tightly coupled memory interface a point-to-point connection.
- Tightly coupled slaves have a data width of 32 bits. Tightly coupled memory interfaces do not support dynamic bus sizing.
- Tightly coupled slaves have a read latency of one cycle, a write latency of zero cycles, and no wait states.

When tightly coupled memory is present, the Nios II core decodes addresses internally to determine if the requested instructions or data reside in tightly coupled memory. If the address resides in tightly coupled memory, the Nios II core accesses the instruction or data through the tightly coupled memory interface. Accessing tightly coupled memory bypasses cache memory. The processor core functions as if cache were not present for the address span of the tightly coupled memory. Instructions for managing the cache do not affect the tightly coupled memory, even if the instruction specifies an address in the range occupied by a tightly coupled memory.

7.5.4.2. Dual Port Memories

Each tightly coupled master connects to one tightly coupled slave over a tightly coupled interface. For this reason, it is helpful to use dual port memories with the tightly coupled instruction master as shown in Figure 265 on page 485. The tightly coupled instruction master is incapable of performing writes because it accesses code for execution only. Without a second memory port connected to an Avalon Memory-Mapped (Avalon-MM) data master, the system does not have write access to the tightly coupled instruction memory. Without write access, code cannot be downloaded into the tightly coupled memory by the Nios II SBT for Eclipse, which makes development and debugging difficult. Without a second port on the tightly coupled

instruction memory, no data master has access to the memory, which means you have no way to view the contents. By making the tightly coupled instruction memory dual port, the embedded processor's data master can be connected to the second port, allowing both reading and writing of data.

7.5.5. Building a Nios II System with Tightly Coupled Memory

This section provides a detailed list of instructions to create a Nios II system in Platform Designer that uses two tightly coupled memories, one for instruction access and one for data access. These two tightly coupled memories are connected to the Nios II processor as shown in Figure 265 on page 485. Additionally, instructions are provided to build a software project to exercise these tightly coupled memories. The output of the software shows that the tightly coupled memories have much faster access times than other on-chip memories.

To build a Nios II system with tightly coupled memory, perform the following steps. These steps are described more fully in the following sections.

1. Modify an existing reference design to include tightly coupled memories.
2. Create the tightly coupled memories in Platform Designer.
3. Connect the tightly coupled memories to the masters.
4. Position the tightly coupled memories in the Nios II processor's address map.
5. Specify the Nios II exception address to access tightly coupled instruction memory.
6. Add a performance counter.
7. Generate the hardware system.
8. Create a software project to exercise the tightly coupled memories.
9. Execute the software on the new hardware design.
10. Change the Tcl scripts and recompile the design to review how the timer settings work.

7.5.5.1. Hardware and Software Requirements

The following hardware and software are required to perform this exercise:

- Nios II development tools version 11.0 or later
- Quartus Prime software
- One of the following Intel development kit boards:
  - Nios II Embedded Evaluation Kit (NEEK), Cyclone® III Edition
  - Embedded Systems Development Kit (ESDK), Cyclone III Edition
  - Stratix® IV GX FPGA Development Kit

7.5.5.2. Modify the Example Design to Include Tightly Coupled Memories

First, create a new hardware reference design with tightly coupled memories that is based on the Nios II Ethernet standard design example. To create this modified reference design, perform the following steps:

1. Navigate to the Nios II Ethernet Standard Design Example web page and locate the Nios II Ethernet Standard design example **.zip** file that corresponds to your board.
2. Extract the files from the downloaded **.zip** file and copy the **niosii_ethernet_standard_**<*board*> directory to a new directory named **standard_tcm**.
3. In the Windows Start menu, choose **Programs** > **Intel FPGA** > **Intel Quartus Prime** <*version*> to run the Intel Quartus Prime software.
4. On the File menu, click **Open Project** and browse to the **standard_tcm \niosii_ethernet_standard_**<*board*>**.qpf** project file. Click **Open**.
5. On the Tools menu, click **Platform Designer**. When prompted, select **eth_std_main_system.qsys** and click **Open** to open the Platform Designer design.
6. Double-click the **cpu** component in the list of available components on the **System Contents** tab to open the Nios II processor configuration wizard.
7. Ensure that **Nios II/f** is selected on the **Core Nios II** tab.
8. Click the **Caches and Memory Interfaces** tab.
9. Select the number of instruction master ports in the drop-down list next to **Number of tightly coupled instruction master port(s)**. In this example, select **1** port.
10. Change the **Instruction cache** to 4 KB.
11. Select the number of instruction master ports in the drop-down list next to **Number of tightly coupled data master port(s)**. In this example, select 1 port.
12. Change the **Data cache** to 2 KB and **Data Cache Line Size** to 4 Bytes.
13. Click **Finish** to close the Nios II processor configuration wizard.

Two new master ports now appear under the **cpu** component called **tightly_coupled_instruction_master_0** and **tightly_coupled_data_master_0**. These master ports are not yet connected to the slave ports.

7.5.5.3. Create the Tightly Coupled Memories

In this section you create two types of tightly coupled memories: a tightly coupled instruction memory and a tightly coupled data memory.

1. In the **Component Library** tab, double-click **On-Chip Memory** in the **On-Chip** subfolder of the **Memories and Memory Controllers** folder. The On-Chip Memory configuration wizard appears.
2. To complete the configuration of this memory, specify the settings listed in the table below.
3. In the Size and Memory initialization boxes, specify the settings listed in the table below.

Table 53. On-Chip Memory Default Settings

Properties		Configuration Settings
Memory type	RAM	Turn this option on
	Dual-Port Access	Turn this option on
	Read During Write Mode	This option is **DONT_CARE**
	Block type	Auto
Size	Memory Width	32
	Total memory size	4 Kb
Read latency	Slave s1	1
	Slave s2	1
Memory initialization	Initialize memory content	Turn this option on
	Enable non-default initialization file	Leave this option off
	Enable In-System Memory Content Editor feature	Leave this option off

4. Click **Finish** to close the configuration wizard.
5. Click the **System Contents** tab and scroll down to the **onchip_memory2_0** component.

   Note: You must name the components exactly as they appear in this tutorial. If your component names differ from the names printed here, the software example will not work.
6. Right-click **onchip_memory2_0** and rename the component to **tightly_coupled_instruction_memory**.
7. To configure a second on-chip memory, in the list of available memory components, double-click **On-Chip Memory (RAM or ROM)**. The On-Chip Memory configuration wizard appears.
8. Configure the settings that are listed in the table below. Unlike the tightly coupled instruction memory, this memory is single-port. Total memory size for tightly coupled data memory is twice the size of tightly coupled instruction memory at 8 KB.

Table 54. On-Chip Memory Default Settings

Properties		Configuration Settings
Memory type	RAM	Turn this option on
	Dual-Port Access	Turn this option off
	Read During Write Mode	This option is **DONT_CARE**
	Block type	Auto
Size	Memory Width	32
	Total memory size	8 Kb
Read latency	Slave s1	1

continued...

Properties		Configuration Settings
Memory initialization	Initialize memory content	Turn this option off
	Enable non-default initialization file	Leave this option off
	Enable In-System Memory Content Editor feature	Leave this option off

9. Click **Finish** to close the On-Chip Memory configuration wizard.
10. Rename the **onchip_memory2_0** component to **tightly_coupled_data_memory**.

7.5.5.4. Connect and Position the Tightly Coupled Memories

To associate the masters with the tightly coupled memories, perform the following steps:

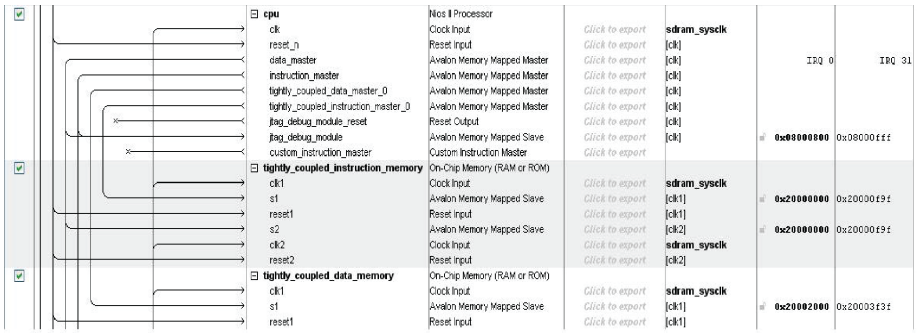
1. To facilitate creating connections between the tightly coupled memory and the Nios II processor, click each new tightly coupled memory and click **Move Up** to move the individual memories just below the cpu component.
2. If necessary, click the + to expand the **tightly_coupled_instruction_memory** component.
3. Using the connections panel in Platform Designer, connect the s1 port of **tightly_coupled_instruction_memory** to the **tightly_coupled_instruction_master_0** component listed under the cpu component. To connect a port, click the empty dot at the intersection of the s1 port and the port you want to connect.
4. Similarly, connect the s2 port of **tightly_coupled_instruction_memory** to the cpu `data_master` port. This connection is shown in Figure 265 on page 485 as the Avalon-MM connection between the Avalon-MM data master port and the Avalon-MM slave port on the tightly coupled instruction memory. Port s2 of this dual-port memory is an Avalon-MM slave port, not a tightly coupled slave port because s2 connects to an Avalon-MM master, which is not a tightly coupled master.
5. If necessary, click the + to expand the **tightly_coupled_data_memory** component.
6. Connect the s1 port of **tightly_coupled_data_memory** to **tightly_coupled_data_master_0**.
7. To change the tightly coupled memories to the same clock domain as the cpu, follow these steps:
   a. Click in the **Clock** column next to `clk1` and `clk2` for the s1 and s2 ports, respectively, of **tightly_coupled_instruction_memory**. A list of available clock signals appears.
   b. Select the same clock domain as the cpu from the list of available clocks to connect this clock to the slave ports.
   c. Similarly, connect `reset1` and `reset2` for the s1 and s2 ports, respectively, to `clk_reset` of the clock source.
   d. Repeat steps a-c for **tightly_coupled_data_memory**.
8. In the **Base** column, enter the base addresses in the table below for all tightly coupled memories.

Table 55. Base Addresses for Tightly Coupled Memories

Port	Base
tightly_coupled_instruction_memory s1	0x20000000
tightly_coupled_instruction_memory s2	0x20000000
tightly_coupled_data_memory s2	0x20002000

The **end** addresses automatically update to reflect the memory size that you specified in the configuration wizard. The base address specification is important. Tightly coupled memories must be mapped so that their addresses do not overlap with the embedded processor's memories and peripherals that are connected to its Avalon-MM instruction and data masters.

Figure 266. Connections for Tightly Coupled Memories



To simplify address decoding, map the high-order address bit to a unique location. By limiting the decoding logic to one bit, you minimize the effect of address decoding on fMAX. The Nios II component works correctly even if the address map is not optimal; however, it displays a warning during system generation.

As an example of optimal address mapping, if all the normal memories and peripherals in your system occupy addresses below 0x2000000, mapping your tightly coupled memories at addresses from 0x2000000 and above satisfies this requirement.

9. To set the exception address of the Nios II processor, open the Nios II processor's configuration wizard. On the **Core Nios II** tab, in the **Exception vector memory:** list, select **tightly_coupled_instruction_memory_s1**.

   The address **Offset** fields are populated automatically and are indexed from the base address specified for the memory module on the **System Contents** tab.

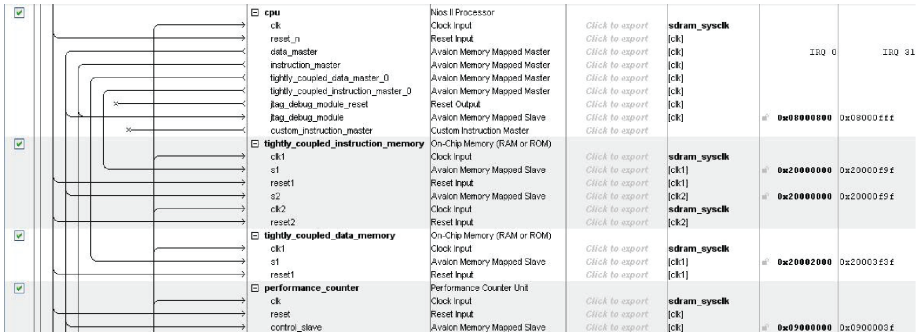
7.5.5.5. Add a Performance Counter

Add a performance counter peripheral so you can compare the performance of the reads and writes to tightly coupled memory to other memories. To add the performance counter, complete the following steps:

1. In the **Component Library** tab, click **Peripherals** to expand the list of available components.
2. Under **Debug and Performance**, double-click **Performance Counter Unit** to open the Performance Counter configuration wizard.
3. Click **Finish**, accepting the default setting of the **3** simultaneously-measured sections.

4. Be sure that the component is named **performance_counter**.
5. Move the component up just below the tightly coupled memory components. Connect the `control_slave` port of the **performance_counter** to the cpu's `data_master` port. The figure below shows these connections.
6. In **performance_counter**, connect `clk` to the same clock domain as cpu and reset to `clk_reset` of the clock source.
7. In the **Base** column, change the base address of **performance_counter** to 0x09000000.

Figure 267. Connections for the Performance Counter



7.5.6. Generate the Platform Designer System

To generate and compile the hardware system, perform the following steps:

1. In the **Generation** tab, click **Generate**. Click **Save** to save the changes in the Platform Designer system.
2. In the Intel Quartus Prime software, on the **Processing** menu, click **Start Compilation**.
3. When the Intel Quartus Prime software compilation is finished, on the Tools menu, click **Programmer** to program the newly generated **niosii_ethernet_standard_**<*board*>**.sof** into the FPGA.

7.5.7. Run the Tightly Coupled Memories Examples from the Nios II Command

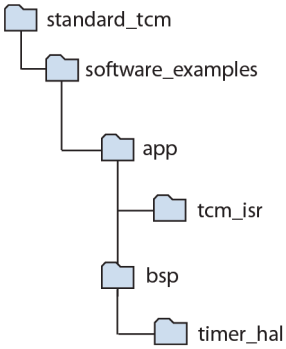
To run the tightly coupled memories example from the Nios II command shell, perform the following steps:

1. To open a Nios II command shell under Windows, in the Start menu, point to Programs > Intel FPGA > Nios II EDS <version number>, and click Nios II <version number> Command Shell.
2. Navigate to the working directory for your project. The following steps refer to this directory as <project_directory>.
3. Ensure that the working directory and all subdirectories are writable by typing the following command:

```
chmod -R +w .
```

4. Download the tcm.zip file and unzip it into the <*project_directory*> directory. The figure below shows the directory structure for the unzipped files.

Figure 268. Project Directory after Unzipping the Files



5. Change to the software_examples/app/tcm_isr subdirectory of your <project_directory> by typing the following command:

```
cd software_examples/app/tcm_isr
```

6. Create and build the application by typing the following command:

```
./create-this-app
```

7. The linker script file, **linker.x** in the **bsp/timer_hal** directory includes a new `isrs_region` located in tightly coupled instruction memory which is adjacent to the `tightly_coupled_instruction_memory` region. isrs_region Listing in linker.x File on page 493 shows the new region.
8. The **tcmisr.objdump** in the **app/tcm_isr** directory defines the `.isrs` section located in the tightly coupled instruction memory. isrs Section Listing in tcm_isr.objdump File on page 493 part of this file.

Example 34. isrs_region Listing in linker.x File

```
MEMORY
{
:
:
tightly_coupled_instruction_memory : ORIGIN = 0x20000120, LENGTH = 1664
timer_isrs_region : ORIGIN = 0x200007a0, LENGTH = 2048
:
:
}
```

Example 35. isrs Section Listing in tcm_isr.objdump File

```
Sections:
Idx Name     Size     VMA      LMA     File off    Algn
0 .entry     00000000  00000000  00000000 00001000    2**5
            CONTENTS, ALLOC, LOAD, READONLY, CODE

1 .exceptions 000001a0  20000120  20000120 00013120    2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE

2 .isrs       000000c4  200007a0  200007a0 000137a0    2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE

3 .text       0000fea4  10000000  10000000 00001000    2**2
              CONTENTS, ALLOC, LOAD, READONLY, CODE

4 .rodata     00000650  1000fea4  1000fea4 00010ea4    2**2
              CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
5 .rwdata     00001ba0  100104f4  100104f4 000114f4    2**2
              CONTENTS, ALLOC, LOAD, DATA, SMALL_DATA

6 .bss        00000154  10012094  10012094 00013094    2**2
              ALLOC, SMALL_DATA
```

7.5.8. Program and Run the Tightly Coupled Memory Project

To program and run a tightly coupled memory project, perform the following steps:

1. To open a terminal session to capture messages from the Nios II processor, type the following command:

   ```
   nios2-terminal
   ```

   If your development board includes more than one JTAG cable, you must specify which cable you are communicating with as an argument to the nios2-terminal command. To do so, first type the following command:

   ```
   jtagconfig -r
   ```

Figure 269. jtagconfig Output

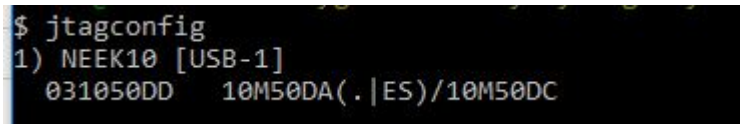


Figure 269 on page 494 shows some sample output from the jtagconfig command. This output shows that the active JTAG cable is number 1. Substitute the number of your JTAG cable for the <cable_number> variable in the following command:

```
nios2-terminal -c <cable_number>
```

2. In your first shell, enter the following command if you have a single JTAG cable:

   ```
   nios2-download -g tcm_isr.elf
   ```

   For development boards with more than one JTAG cable, enter the following command:

   ```
   nios2-download -c <cable_number> -g tcm_isr.elf
   ```

   Figure 270 on page 495 is a printout of the statistics that shows the higher speeds obtained by leveraging tightly coupled memories. Note that the number of clock cycles for tightly coupled memory is very similar to that of the cached memory. The result demonstrates that tightly coupled memories allow fixed low-latency read access to executable code as well as fixed low-latency read, write, or read and write access to data.

   Note: The timing numbers output varies between Nios development boards.

Figure 270. Tightly Coupled Memory versus Cache Example Real-Time Measures

```
Tightly Coupled Memory vs. Cache Example Real-Time Measurements:

MAX 10 (q_sys).
Interrupt response time:           193 clock cycles.

Checksum Times:                          All 3 memories match.
Tightly coupled memory:     74.12 microseconds,     3706 clocks.
SDRAM memory:              110.62 microseconds,     5531 clocks.
SDRAM memory (cached):      74.12 microseconds,     3706 clocks.

Checksum loop measured iteration:  3.
Checksum value:                  34465 total.
Checksum block size:               320 words (32 bits).
CPU Frequency:                 50.0000 Mhz.
```

7.5.9. Understanding the Tcl Scripts

The following sections describe creating special memory regions for the timer memory interrupt service routines and timer definitions and locating the exception stack in the tightly coupled memory.

7.5.9.1. Timer Memory

The **timer_memory_section.tcl** script is located in the **bsp/timer_hal** directory. This Tcl script reserves 2048 bytes of the tightly coupled instruction memory. The reserved space is used to store the timer interrupt service routines.

The **timer_memory_section.tcl** script takes the `tightly_coupled_instruction` memory region and separates out 2048 bytes of the memory region into a new region called `timer_isr_region`. The next line of code adds a section mapping the `.isrs` section to the `timer_isr_region`.

```
# Create tightly_coupled_memory region.
add_memory_region tightly_coupled_instruction_memory $slave $offset $new_span
# Create a second region called timer_isr memory_region.
add_memory_region timer_isrs_region $slave $split_offset $split_span
# Create memory mapping to map .isrs to timer_isrs_region.
add_section_mapping .isrs timer_isrs_region
```

The **timer_interrupt_latency.h** file is also updated to reflect the change in the section mapping of the `timer_interrupt_latency_irq()` timer interrupt service routine to `.isrs` instead of `.exceptions`. The timer interrupt service routines are now stored in the `timer_isr_region`.

The interrupt service routines must be located in the new `.isrs` section. Otherwise, the linker uses the default setting, defeating the purpose of declaring a special memory section for the interrupt service routine.

To locate the interrupt service routines in the new .isrs section, complete the following steps:

1. Add a section mapping to map the `.isrs` section to the newly added memory region.
2. Edit your source files to ensure the interrupt service routines are mapped to the new memory section.
3. Compile your project files and check the **linker.x** and **tcm_isr.objdump** files to ensure that the section mapping and memory regions are declared correctly and contain the interrupt service routine.

Note: For more information about linker memory regions, refer to the Nios II Software Build Tools chapter in the *Nios II Software Developer's Handbook*.

7.5.9.2. Exception Stack

The **timer_memory_section.tcl** script locates the exception stack in the `tightly_coupled_data_memory` region.

Example 36. Exception Stack

```
# Locate the exception stack to tightly coupled data memory.
set_setting hal.linker.enable_exception_stack TRUE
set_setting hal.linker.exception_stack_memory_region_name
tightly_coupled_data_memory.
aset_setting hal.linker.exception_stack_size 1024
```

7.5.9.3. Timer Definitions

The following sections describe `peripheral_subsystem_sys_clk_timer` and `peripheral_subsystem_high_res_timer`.

7.5.9.3.1. peripheral_subsystem_sys_clk_timer

The **timer_definition.tcl** script is located in the **bsp/timer_hal** directory. The script defines the timers as follows:

```
set_setting hal.sys_clk_timer peripheral_subsystem_sys_clk_timer
set_setting hal.timestamp_timer none
```

This script is essential for the clocks definitions. The software driver `hal.sys_clk_timer` must be driven by the hardware clock named `peripheral_subsystem_sys_clk_timer`. Connecting `hal.sys_clk_timer` to any other hardware timer results in a compilation error. The following exercise demonstrates this point.

1. Delete or rename the **Makefile** in the **app/tcm_isr** folder. Then delete or rename **public.mk** from the **bsp/timer_hal** folder.
2. Open the **timer_definition.tcl** file and change `peripheral_subsystem_sys_clk_timer` to `peripheral_subsystem_high_res_timer` as follows:

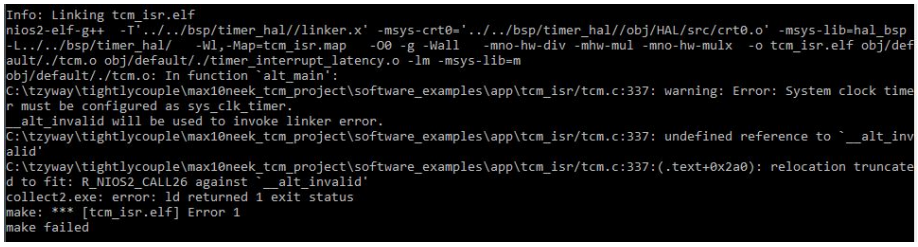
```
set_setting hal.sys_clk_timer peripheral_subsystem_high_res_timer
set_setting hal.timestamp_timer none
```

3. Save **timer_definition.tcl**.
4. Return to your shell and recreate the application by typing:

```
./create-this-app
```

5. The figure below shows the error. Setting hal.sys_clk_timer to any other timers except for peripheral_subsystem_sys_clk_timer results in the same error message.

Figure 271. Error Message after Changing the hal.sys_clk_timer



7.5.9.3.2. peripheral_subsystem_high_res_timer

The hardware timer called `peripheral_subsystem_high_res_timer` calculates interrupt latency. `timer_interrupt_latency_init()`, defined in **timer_interrupt_latency.c**, installs an interrupt service routine to handle `peripheral_subsystem_high_res_timer`. Therefore, `peripheral_subsystem_high_res_timer` must not be tied to the software timestamp driver, `hal.timestamp_timer`; it is set to none. Because `peripheral_subsystem_sys_clk_timer` is used for `hal.sys_clk_timer`, it must not be used for `hal.timestamp_timer`.

The following exercise shows this point:

1. If you have not already done so, delete or rename **Makefile** in the **app/tcm_isr** folder. Delete or rename **public.mk** in the **bsp/timer_hal** folder.
2. Open the **timer_definition.tcl** file and change the setting of `hal.timestamp_timer` from none to `peripheral_subsystem_high_res_timer` as follows:

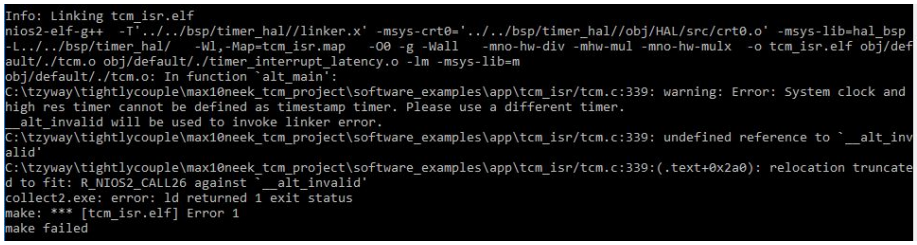
```
set_setting hal.sys_clk_timer peripheral_subsystem_sys_clk_timer
set_setting hal.timestamp_timer peripheral_subsystem_high_res_timer
```

3. Save **timer_definition.tcl**.
4. Return to your shell and recreate the application by typing the following command:

```
./create-this-app
```

5. The figure below shows the error that you get. Setting hal.timestamp_timer to peripheral_subsystem_sys_clk_timer or peripheral_subsystem_high_res_timer results in the same error message. Setting hal.timestamp_timer to other hardware timers in the system will not result in the error message in Figure 8.

Figure 272. Error Message after Changing the hal.timestamp_timer



7.6. Optimizing Nios II Based Systems and Software Revision History

Document Version	Changes
2020.07.22	• Added information about using the Modular Scatter-Gather DMA (mSGDMA) Core in section: *Improving Data Movement*. • Added a new section: *Accessing Multiple Nios II Designs*.
2017.06.12	Added a new section: *Using Tightly Coupled Memory with the Processor Tutorial.*
2016.12.19	Initial release.

Related Information

Document Revision History for Embedded Design Handbook on page 6