Embedded Design Handbook

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

ED_HANDBOOK-2.2

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ED_HANDBOOK-2.2

Contents

Chapter Revision Dates

Section I. Introduction

Chapter 1. First Time Designer's Guide

Introduction 1-1
First Time Designer's Guide Introduction 1-1
FPGAs and Soft-Core Processors 1-1
Embedded System Design 1-2
FPGA Hardware Design 1-2
Connecting Your FPGA Design to Your Board 1-4
Connecting Signals to your SOPC Builder System 1-4
Constraining Your FPGA-Based Design 1-5
SOPC Builder Design 1-5
Design Replication 1-6
Customization and Acceleration 1-7
Software Design 1-8
Tools Description 1-8
Nios II IDE Flow 1-8
Software Build Tools Flow 1-11
Board Design Considerations 1-11
Configuration 1-12
Booting 1-13
Additional Design Considerations 1-14
Resources 1-14
Support 1-15
Training 1-15
Documentation 1-15
Third Party Intellectual Property 1-16
Glossary 1-16
Conclusion 1-17
Referenced Documents 1-17
Document Revision History 1-18

Section II. Nios II Software Development

Chapter 2. Developing Nios II Software

Introduction 2-1
Software Development Cycle 2-2
Altera System on a Programmable Chip (SOPC) Solutions 2-2
Nios II Software Development Process 2-3
Software Project Mechanics 2-5
Software Tools Background 2-5
Development Flow Guidelines 2-6
Nios II Software Build Tools Flow 2-6
Configuring BSP and Application Projects 2-7
Software Example Designs 2-7

Configuring the BSP Project 2-7
Configuring the Application Project 2-9
Software Project Development Mechanics 2-10
Ensuring Software Project Coherency 2-12
Developing With the Hardware Application Layer 2-16
Overview of the HAL 2-16
HAL Configuration Options 2-16
System Startup in HAL-Based Applications 2-17
System Initialization 2-17
crt0 Initialization 2-18
HAL Initialization 2-19
HAL Peripheral Services 2-20
Timers 2-20
Character Mode Devices 2-22
Flash Memory Devices 2-24
Direct Memory Access (DMA) Devices 2-26
Files and File Systems 2-29
Ethernet Devices 2-30
Unsupported Devices 2-30
Accessing Memory With the Nios II Processor 2-31
Creating General C/C++ Applications 2-31
Accessing Peripherals 2-31
Sharing Uncached Memory 2-32
Sharing Memory With Cache Performance Benefits 2-32
Handling Exceptions 2-33
Modifying the Exception Handler 2-34
Optimizing the Application 2-34
Performance Tuning Background 2-35
Speeding Up System Processing Tasks 2-35
Analyzing the Problem 2-35
Accelerating your Application 2-35
Accelerating Interrupt Service Routines 2-38
Analyzing the Problem 2-38
Accelerating the Interrupt Service Routine 2-38
Reducing Code Size 2-38
Analyzing the Problem 2-38
Reducing the Code Footprint 2-39
Linking Applications 2-40
Background 2-40
Linker Sections and Application Configuration 2-40
HAL Linking Behavior 2-40
Default BSP Linking 2-41
User-Controlled BSP Linking 2-41
Application Boot Loading and Programming System Memory 2-42
Default BSP Boot Loading Configuration 2-43
Boot Configuration Options 2-43
Booting and Running From Flash Memory 2-44
Booting From Flash Memory and Running From Volatile Memory 2-45
Booting and Running From Volatile Memory 2-45
Booting From Altera EPCS Memory and Running From Volatile Memory 2-46
Booting and Running From FPGA Memory 2-46
Generating and Programming System Memory Images 2-47
Programming FPGA Memory 2-47
Configuring and Programming Flash Memory 2-47

Conclusion 2-49
Referenced Documents 2-49
Document Revision History 2-50

Chapter 3. Debugging Nios II Designs

Debuggers 3-1
Nios II Software Development Tools 3-1
Nios II System ID 3-2
Project Templates 3-3
Configuration Options 3-3
Nios II GDB Console and GDB Commands 3-5
Nios II Terminal Window and stdio Library Functions 3-6
Importing Projects Created Using the Nios II Software Build Tools 3-7
Selecting a Processor Instance in a Multiple Processor Design 3-7
FS2 Console 3-9
SignalTap II Embedded Logic Analyzer 3-10
Lauterbach Trace32 Debugger and PowerTrace Hardware 3-10
Debugging the Lauterbach PowerTrace to Nios II Processor Connection 3-10
C Source Correlation 3-11
Registering Trace Signals 3-11
Insight and Data Display Debuggers 3-11
Run-Time Analysis Debug Techniques 3-11
Software Profiling 3-11
Watchpoints 3-12
Stack Overflow 3-13
Hardware Abstraction Layer (HAL) 3-13
Breakpoints 3-13
Debugger Stepping and Using No Optimizations 3-14
Conclusion 3-15
Referenced Documents 3-15
Document Revision History 3-15

Chapter 4. Nios II Command-Line Tools

Introduction 4-1
Altera Command-Line Tools for Board Bringup and Diagnostics 4-1
jtagconfig 4-1
jtagconfig Usage Example 4-2
nios2-configure-sof 4-3
nios2-configure-sof Usage Example 4-3
system-console 4-3
Altera Command-Line Tools for Hardware Development 4-4
quartus_cmd and sopc_builder 4-4
Altera Command-Line Tools for Flash Programming 4-6
nios2-flash-programmer 4-6
nios2-flash-programmer Usage Example 4-6
elf2flash, bin2flash, and sof2flash 4-7
bin2flash Usage Example 4-8
Altera Command-Line Tools for Software Development and Debug 4-8
nios2-terminal 4-9
nios2-download 4-9
nios2-download Usage Example 4-9
nios2-stackreport 4-9

nios2-stackreport Usage Example . . . 4-10
validate_zip . . . 4-10
validate_zip Usage Example . . . 4-10
nios2-ide . . . 4-10
Linux wrapper script . . . 4-10
Windows wrapper script . . . 4-11
nios2-gdb-server . . . 4-11
nios2-gdb-server Usage Example . . . 4-11
nios2-debug . . . 4-12
nios2-debug Usage Example . . . 4-12
Altera Command-Line Nios II Software Build Tools . . . 4-13
BSP Related Tools . . . 4-13
Application Related Tools . . . 4-14
GNU Command-Line Tools . . . 4-14
nios2-elf-addr2line . . . 4-14
nios2-elf-addr2line Usage Example . . . 4-15
nios2-elf-gdb . . . 4-15
nios2-elf-readelf . . . 4-15
nios2-elf-readelf Usage Example . . . 4-15
nios2-elf-ar . . . 4-16
nios2-elf-ar Usage Example . . . 4-16
Linker . . . 4-16
Linker Usage Example . . . 4-16
nios2-elf-size . . . 4-17
nios2-elf-size Usage Example . . . 4-17
nios2-elf-strings . . . 4-17
nios2-elf-strings Usage Example . . . 4-17
nios2-elf-strip . . . 4-17
nios2-elf-strip Usage Example . . . 4-17
nios2-elf-strip Usage Notes . . . 4-17
nios2-elf-gdbtui . . . 4-18
nios2-elf-gprof . . . 4-18
nios2-elf-insight . . . 4-18
nios2-elf-gcc and g++ . . . 4-18
Compilation Command Usage Example . . . 4-18
More Complex Compilation Example . . . 4-19
nios2-elf-c++filt . . . 4-19
nios2-elf-c++filt Usage Example . . . 4-19
More Complex nios2-elf-c++filt Example . . . 4-19
nios2-elf-nm . . . 4-20
nios2-elf-nm Usage Example . . . 4-20
More Complex nios2-elf-nm Example . . . 4-20
nios2-elf-objcopy . . . 4-20
nios2-elf-objcopy Usage Example . . . 4-21
nios2-elf-objdump . . . 4-21
nios2-elf-objdump Usage Description . . . 4-21
nios2-elf-ranlib . . . 4-21
Referenced Documents . . . 4-21
Document Revision History . . . 4-22

Chapter 5. Optimizing Nios II C2H Compiler Results

Introduction . . . 5-1
Prerequisites . . . 5-1
Cost and Performance . . . 5-1

Overview of the C2H Optimization Process 5-2
Getting Started 5-2
Iterative Optimization 5-3
Meeting Your Cost and Performance Goals 5-3
Factors Affecting C2H Results 5-3
Memory Accesses and Variables 5-4
Arithmetic and Logical Operations 5-5
Statements 5-6
Control Flow 5-7
If Statements 5-7
Loops 5-8
Subfunction Calls 5-8
Resource Sharing 5-9
Data Dependencies 5-9
Memory Architecture 5-10
Data Cache Coherency 5-11
DRAM Architecture 5-11
Efficiency Metrics 5-11
Cycles Per Loop Iteration (CPLI) 5-12
FPGA Resource Usage 5-12
Avalon-MM Master Ports 5-13
Embedded Multipliers 5-13
Embedded Memory 5-14
Data Throughput 5-14
Optimization Techniques 5-14
Pipelining Calculations 5-14
Increasing Memory Efficiency 5-16
Use Wide Memory Accesses 5-16
Segment the Memory Architecture 5-18
Use Localized Data 5-19
Reducing Data Dependencies 5-20
Use __restrict__ 5-20
Reducing Logic Utilization 5-24
Use "do-while" rather than "while" 5-24
Use Constants 5-25
Leave Loops Rolled Up 5-27
Use ++ to Sequentially Access Arrays 5-28
Avoid Excessive Pointer Dereferences 5-28
Avoid Multipliers 5-28
Avoid Arbitrary Division 5-29
Use Masks 5-31
Use Powers of Two in Multi-Dimensional Arrays 5-31
Use Narrow Local Variables 5-32
Optimizing Memory Connections 5-32
Remove Unnecessary Connections to Memory Slave ports 5-32
Reduce Avalon-MM Interconnect Using #pragma 5-33
Remove Unnecessary Memory Connections to Nios II Processor 5-35
Optimizing Frequency Versus Latency 5-35
Improve Conditional Latency 5-36
Improve Conditional Frequency 5-37
Improve Throughput 5-38
Avoid Short Nested Loops 5-38
Remove In-place Calculations 5-39
Replace Arrays 5-41

Use Polled Accelerators 5-42
Use an Interrupt-Based Accelerator 5-43
Glossary 5-43
Referenced Documents 5-45
Document Revision History 5-45

Section III. System-Level Design

Chapter 6. Avalon Memory-Mapped Design Optimizations

Selecting Hardware Architecture 6-1
Bus 6-2
Full Crossbar Switch 6-2
Partial Crossbar Switch 6-3
Streaming 6-5
Dynamic Bus Sizing 6-6
Understanding Concurrency 6-7
Create Multiple Masters 6-7
Create Separate Datapaths 6-8
Use DMA Engines 6-8
Include Multiple Master or Slave Ports 6-9
Create Separate Sub-Systems 6-10
Increasing Transfer Throughput 6-12
Using Pipelined Transfers 6-12
Maximum Pending Reads 6-13
Selecting the Maximum Pending Reads Value 6-13
Overestimating Versus Underestimating the Maximum Pending Reads Value 6-13
Pipelined Read Masters 6-13
Requirements 6-14
Throughput Improvement 6-14
Pipelined Read Master Example 6-15
Arbitration Shares and Bursts 6-16
Differences between Arbitration Shares and Bursts 6-16
Choosing Interface Types 6-17
Burst Master Example 6-18
Increasing System Frequency 6-20
Use Pipeline Bridges 6-20
Master-to-Slave Pipelining 6-20
Slave-to-Master Pipelining 6-21
waitrequest Pipelining 6-21
Use a Clock Crossing Bridge 6-22
Increasing Component Frequencies 6-22
Reducing Low-Priority Component Frequencies 6-22
Consequences of Using Bridges 6-23
Increased Latency 6-23
Limited Concurrency 6-25
Address Space Translation 6-27
Address Shifting 6-27
Address Coherency 6-28
Minimize System Interconnect Logic 6-29
Use Unique Address Bits 6-29
Create Dedicated Master and Slave Connections 6-30
Remove Unnecessary Connections 6-30

Reducing Logic Utilization 6-30
Minimize Arbitration Logic by Consolidating Components 6-30
Logic Consolidation Tradeoffs 6-31
Combined Component Example 6-31
Use Bridges to Minimize System Interconnect Fabric Logic 6-32
SOPC Builder Speed Optimizations 6-33
Reduced Concurrency 6-34
Use Bridges to Minimize Adapter Logic 6-35
Effective Placement of Bridges 6-35
Compact System Example 6-35
Reducing Power Utilization 6-37
Reduce Clock Speeds of Non-Critical Logic 6-37
Clock Crossing Bridge 6-38
Clock Crossing Adapter 6-39
Minimize Toggle Rates 6-40
Registering Component Boundaries 6-40
Enabling Clocks 6-41
Inserting Bridges 6-41
Disable Logic 6-41
Software Controlled Sleep Mode 6-41
Hardware Controlled Sleep Mode 6-42
Referenced Documents 6-42
Document Revision History 6-43

Chapter 7. Memory System Design
Overview 7-1
Volatile Memory 7-1
Non-volatile Memory 7-1
On-Chip Memory 7-1
Advantages 7-2
Disadvantages 7-2
Best Applications 7-2
Cache 7-2
Tightly Coupled Memory 7-3
Look Up Tables 7-3
FIFO 7-3
Poor Applications 7-3
On-Chip Memory Types 7-3
Best Practices 7-3
External SRAM 7-4
Advantages 7-4
Disadvantages 7-4
Best Applications 7-5
Poor Applications 7-5
External SRAM Types 7-5
Best Practices 7-5
Flash 7-6
Advantages 7-6
Disadvantages 7-6
Typical Applications 7-7
Poor Applications 7-7
Flash Types 7-7

SDRAM 7-8
Advantages 7-9
Disadvantages 7-9
Best Applications 7-9
Poor Applications 7-9
SDRAM Types 7-10
SDRAM Controller Types Available From Altera 7-10
Best Practices 7-11
Half-Rate Mode 7-11
Full-Rate Mode 7-11
Sequential Access 7-12
Bursting 7-12
SDRAM Minimum Frequency 7-12
SDRAM Device Speed 7-12
Memory Optimization 7-13
Isolate Critical Memory Connections 7-13
Match Master and Slave Data Width 7-13
Use Separate Memories to Exploit Concurrency 7-13
Understand the Nios II Instruction Master Address Space 7-14
Test Memory 7-14
Case Study 7-14
Application Description 7-14
Initial Memory Partitioning 7-15
Optimized Memory Partitioning 7-16
Add An External SRAM for input buffers 7-16
Add On-Chip Memory for Video Line Buffers 7-17
Referenced Documents 7-18
Document Revision History 7-19

Chapter 8. Hardware Acceleration and Coprocessing

Hardware Acceleration 8-1
Accelerating Cyclic Redundancy Checking (CRC) 8-1
Matching I/O Bandwidths 8-3
Pipelining Algorithms 8-3
Creating Nios II Custom Instructions 8-4
Using the C2H Compiler 8-7
Coprocessing 8-8
Creating Multicore Designs 8-8
Pre- and Post-Processing 8-10
Replacing State Machines 8-11
Low-Speed State Machines 8-12
High-Speed State Machines 8-13
Subdivided State Machines 8-13
Referenced Documents 8-13
Document Revision History 8-14

Chapter 9. Verification and Board Bring-Up

Introduction 9-1
Verification Methods 9-1
Prerequisites 9-1
FS2 Console 9-2
SOPC Builder Test Integration 9-2

Capabilities of the FS2 Console 9-3
System Console 9-5
SignalTap II Embedded Logic Analyzer 9-7
External Instrumentation 9-8
SignalProbe 9-8
Logic Analyzer Interface 9-9
Stimuli Generation 9-9
Board Bring-up 9-10
Peripheral Testing 9-10
Data Trace Failure 9-11
Address Trace Failure 9-11
Device Isolation 9-12
JTAG 9-13
Board Testing 9-14
Minimal Test System 9-15
System Verification 9-17
Designing with Verification in Mind 9-17
Accelerating Verification 9-18
Using Software to Verify Hardware 9-19
Environmental Testing 9-21
Referenced Documents 9-22
Document Revision History 9-23

Chapter 10. Interfacing an External Processor to an Altera FPGA
Configuration Options 10-2
RapidIO Interface 10-5
PCI Express Interface 10-7
PCI Interface 10-9
PCI Lite Interface 10-9
Serial Protocol Interface (SPI) 10-10
Custom Bridge Interfaces 10-11
Conclusion 10-13
Referenced Documents 10-13
Document Revision History 10-14

Additional Information
How to Contact Altera Info-1
Typographic Conventions Info-1

Chapter Revision Dates

The chapters in this book, the *Embedded Design Handbook*, were revised on the following dates:

Chapter 1. First Time Designer's Guide
Revised: *January 2009*
Part number: *ED51001-2.1*

Chapter 2. Developing Nios II Software
Revised: *June 2008*
Part number: *ED51002-1.1*

Chapter 3. Debugging Nios II Designs
Revised: *June 2008*
Part number: *ED51003-1.1*

Chapter 4. Nios II Command-Line Tools
Revised: *November 2008*
Part number: *ED51004-2.0*

Chapter 5. Optimizing Nios II C2H Compiler Results
Revised: *June 2008*
Part number: *ED51005-1.1*

Chapter 6. Avalon Memory-Mapped Design Optimizations
Revised: *June 2008*
Part number: *ED51007-1.1*

Chapter 7. Memory System Design
Revised: *June 2008*
Part number: *ED51008-1.1*

Chapter 8. Hardware Acceleration and Coprocessing
Revised: *June 2008*
Part number: *ED51006-1.1*

Chapter 9. Verification and Board Bring-Up
Revised: *November 2008*
Part number: *ED51010-1.2*

Chapter 10. Interfacing an External Processor to an Altera FPGA
Revised: *February 2009*
Part number: *ED51011-1.0*

Section I. Introduction

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

This section includes the following chapters:

- Chapter 1, First Time Designer's Guide

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

1. First Time Designer's Guide

ED51001-2.1

Introduction

Altera® provides various tools for development of hardware and software for embedded systems. This handbook complements the primary documentation for these tools by describing how to most effectively use the tools. It recommends design styles and practices for developing, debugging, and optimizing embedded systems using Altera-provided tools. The handbook introduces concepts to new users of Altera's embedded solutions, and helps to increase the design efficiency of the experienced user.

This handbook is not a comprehensive reference guide. For general reference and detailed information, refer to the primary documentation cited in this handbook.

This first chapter of the handbook contains information about the Altera embedded development process and procedures for the first time user. The remaining chapters focus on specific aspects of embedded development for Altera FPGAs.

First Time Designer's Guide Introduction

This chapter is for first time users of Altera's embedded development tools for hardware and software development. The chapter provides information about the design flow and development tools interaction, and describes the differences between the Nios® II processor flow and a typical discrete microcontroller design flow.

However, this chapter does not replace the basic reference material for the first time designer, such as the *Nios II Processor Reference Handbook*, the *Nios II Software Developer's Handbook*, volumes 4 and 5 of the *Quartus II Handbook*, and the *Nios II Flash Programmer's Guide*.

FPGAs and Soft-Core Processors

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

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

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

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

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

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

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

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

Embedded System Design

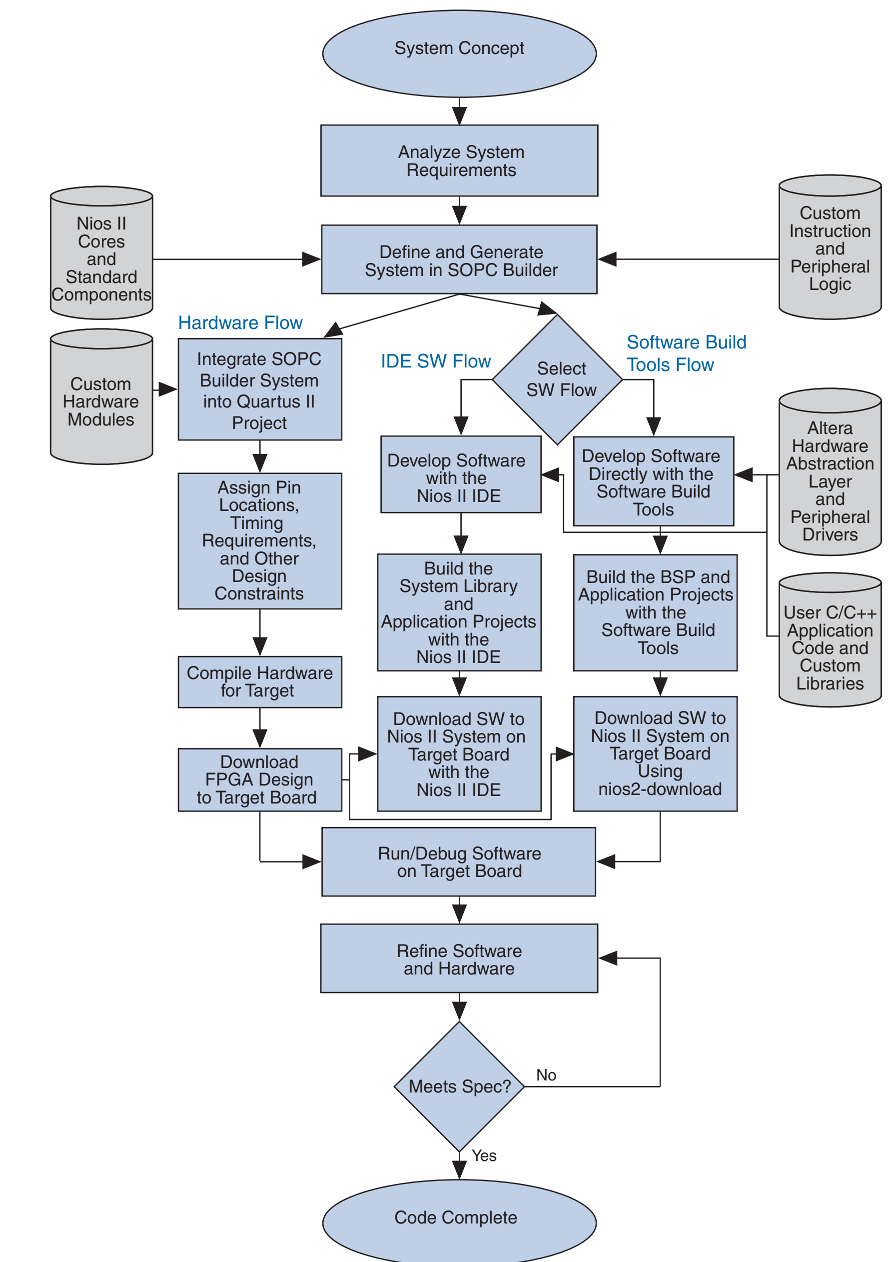
FPGA Hardware Design

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

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

Figure 1–1 illustrates the FPGA hardware design process and Nios II software flow.

Figure 1–1. System Design Flow



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

Connecting Your FPGA Design to Your Board

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

1. Identify the top level of your FPGA design.
2. Assign signals in the top level of your FPGA design to pins on your FPGA using any of the methods mentioned at the Altera I/O Management, Board Development Support, and Signal Integrity Analysis Resource Center, at www.altera.com/support/software/io-board/sof-qts-io.html

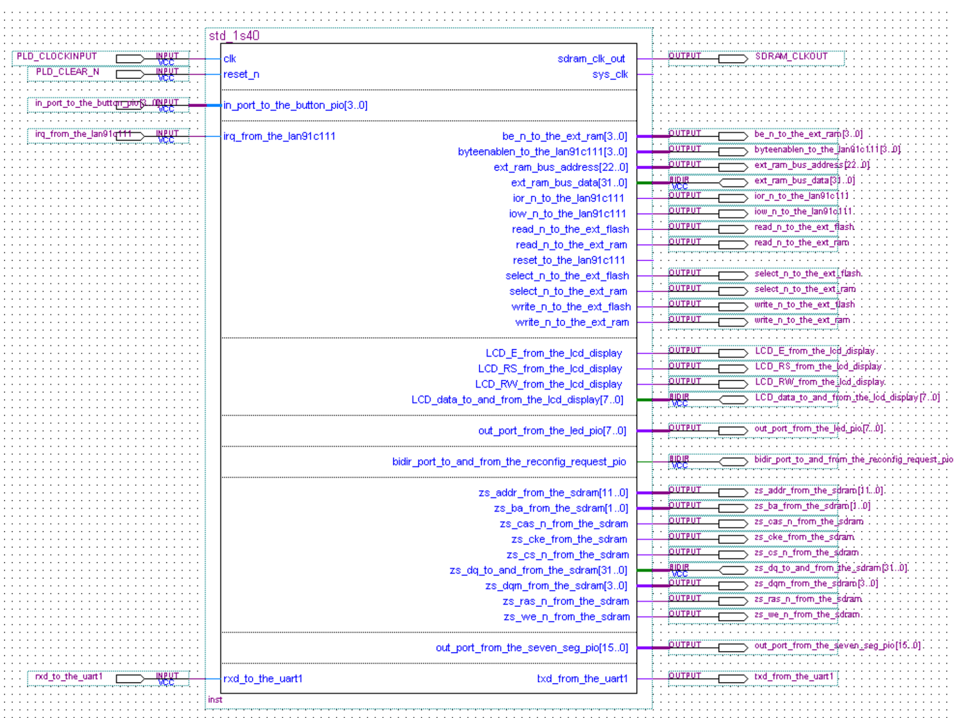
The top level of your FPGA-based design might be your SOPC Builder system. However, the FPGA can include additional design logic.

Connecting Signals to your SOPC Builder System

You must define the clock and reset pins for your SOPC Builder system. You must also define each I/O signal that is required for proper system operation. Figure 1–2 shows the top level block diagram of an SOPC Builder system that includes a Nios II processor. The large symbol in this top-level diagram, labeled **std_1s40**, represents the SOPC Builder system. The flag-shaped pin symbols in this diagram represent off-chip (off-FPGA) connections.

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

Figure 1–2. Top Level Block Diagram



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® II software or by a third party EDA provider. The Quartus II software uses your constraint information during design compilation to achieve Altera's best possible results.

Altera's third-party EDA partners and the tools they provide are listed at www.altera.com/products/software/partners/eda_partners/eda-tools.html

SOPC Builder Design

SOPC Builder simplifies the task of building complex hardware systems on an FPGA. SOPC Builder allows you to describe the topology of your system using a graphical user interface (GUI) and then generate the hardware description language (HDL) files for that system. The Quartus II software compiles the HDL files to create an FPGA programming file.

For additional information about SOPC Builder, refer to *Volume 4: SOPC Builder* of the *Quartus II Handbook*.

SOPC Builder 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 SOPC Builder.

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

For information about how to create pin assignments, refer to the Quartus II online Help and to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Altera recommends that you start your design from a small pretested project and build it incrementally. Start with one of the many SOPC Builder example designs provided with the Nios II Embedded Design Suite (EDS), or with a design example from the *Nios II Hardware Development Tutorial*.

The Nios II EDS includes several SOPC Builder-based hardware example designs and corresponding software examples. The software examples are located in the hardware project directory of your Altera Nios development board type—for example, **$SOPC_KIT_NIOS2\examples\verilog\niosII_cycloneII_2c35**—in the **software_examples** subdirectory for your design type.

For more information about the examples provided in the Nios II EDS, refer to the "Using Nios II Example Design Scripts" section of the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

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

After you run a simple software design—such as the simplest built-in example, Hello World Small—build individual systems based on this design to test the additional interfaces or custom options that your system requires. Altera recommends that you start with a simple system that includes a processor with a JTAG debug module, an on-chip memory component, and a JTAG UART component, and create a new system for each new untested component, rather than adding in new untested components incrementally.

After you verify that each new hardware component functions correctly in its own separate system, you can combine the new components incrementally in a single SOPC Builder system. SOPC Builder 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*.

Design Replication

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

- Quartus II project file (**.qpf**)
- Quartus II settings file (**.qsf**)

  The **.qsf** file contains all of the device, pin, timing, and compilation settings for the Quartus II project.
- A top level design file – schematic (**.bdf**), Verilog HDL (**.v**), or VHDL (**.vhd**)

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

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

  For details about the design file types, refer to the Quartus II online Help.
- Internal SOPC Builder description file (**.sopc**)
- SOPC Builder description file (**.sopcinfo**)

  This file contains an XML description of your SOPC Builder system. SOPC Builder and downstream tools, including the software build tools, derive information about your system from this file.
- Your software application source files

To replicate an entire project (both hardware and software), first copy the required files to a separate directory, and then open the new project. You can open the new project in the Quartus II software, in SOPC Builder, or in the Nios II Integrated Development Environment (IDE). You can also create a script to automate the copying process.

For more information about all of these files, refer to the *Archiving SOPC Builder Projects* chapter in volume 4 of the *Quartus II Handbook*.

Customization and Acceleration

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. SOPC Builder facilitates this flexibility, by allowing you to add and modify system components and regenerate your project easily.

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. SOPC Builder 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
- C2H accelerated software

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.

Working with a software or systems engineer, use the C2H Compiler to help analyze sophisticated algorithms to determine potential hardware acceleration gains. As in any hardware acceleration methodology, you must make trade-offs between performance and resource consumption. When a C compiler compiles code using a high level of optimization, the resulting executable program typically runs faster, but also often consumes more memory than similar code compiled with a lower level of optimization. Similarly, accelerators built with the C2H Compiler typically run faster than the unaccelerated code, but they consume more FPGA resources.

For information about hardware acceleration, refer to the *Hardware Acceleration and Coprocessing* chapter of the *Embedded Design Handbook*. For information about how to use the C2H Compiler, refer to the *Nios II C2H Compiler User Guide* and to the *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*. For information on custom instructions, refer to the *Nios II Custom Instruction User Guide*. For information on creating custom peripherals, refer to the *Developing Components for SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.

Software Design

This section contains brief descriptions of the software design tools provided by the Nios II EDS, the Nios II IDE development flow, and the software build tools development flow.

Tools Description

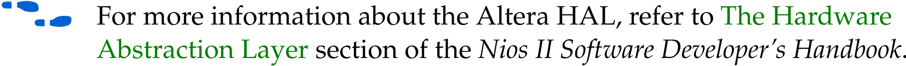
The Nios II EDS provides the following tools for software development:

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

  For an overview of these and other Altera-provided utilities, refer to the *Nios II Command-Line Tools* chapter of the *Embedded Design Handbook*.

- Nios II processor-specific port of the newlib C library
- Hardware abstraction layer (HAL)

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

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

- Nios II IDE

  The Nios II IDE is 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.

- Nios II software build tools flow

  The Nios II software build tools development flow is a scriptable, command-line based development flow that uses the software build tools independent of the Nios II IDE.

  For more information about the Nios II software build tools flow, refer to the *Developing Nios II Software* chapter of the *Embedded Design Handbook*.

Nios II IDE Flow

To learn about the Nios II IDE, refer to the Nios II software development tutorial. Unlike the *Nios II Hardware Development Tutorial*, this tutorial is contained in the Nios II IDE Help system. To open this Help system, in the Nios II IDE, on the Help menu, click **Welcome**. A PDF version is also available at www.altera.com/literature/ug/ug_nios2_ide_help.pdf.

The Nios II software development tutorial teaches you about the following key elements of the flow:

- System library project
- Software abstraction of the SOPC Builder hardware design
- Application project
- The software that drives your application

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

Altera recommends that you use an Altera Nios II 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.

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

Debugging Options

The Nios II EDS provides the following programs to aid in debugging your hardware and software system:

- A built-in Nios II IDE Debugger
- Several distinct interfaces to the GNU Debugger (GDB)
- A Nios II-specific implementation of the First Silicon Solutions, Inc. FS2 console (available on Windows platforms only)
- System Console, a system debug console

You can begin debugging software immediately using the built-in Nios II IDE 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 IDE)
- 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*.

For detailed information about the FS2 console, refer to the documentation in the **$SOPC_KIT_NIOS2\bin\fs2\doc** directory and to the *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook*.

The System Console is a system debug console that provides the SOPC Builder designer with a Tcl-based, scriptable command-line interface for performing system or individual component testing. It is available in Nios II EDS version 8.0 and later.

For detailed information about the System Console, refer to the *System Console User Guide*. On-line training is available at http://www.altera.com/training.

Third party debugging environments are also available from vendors such as Lauterbach Datentechnik GmBH and First Silicon Solutions, Inc.

Command Line

You can use the Nios II IDE to create your project. The Nios II IDE guides you if you are unfamiliar with the Nios II software toolchain. It also provides easy access to newlib library functions and the HAL software layer.

However, some actions, such as rebuilding software after minor source code edits, do not require the IDE. In these cases, you may rebuild the project from a Nios II command shell, using your application's makefile. For example, to build or rebuild your software, perform the following steps:

1. Open a Nios II command shell.

   To start the Nios II command shell on Windows platforms, on the Start menu, click **All Programs**. On the All Programs menu, on the Altera submenu, on the Nios II EDS <*version*> submenu, click **Nios II** <*version*> **Command Shell**.
   On Linux platforms, type the following command:
   `$SOPC_KIT_NIOS2/sdk_shell` ↵

2. Change to the directory in which your makefile is located. If you use the Nios II IDE 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` ↵

Example 1–1 illustrates the output of the `make` command run on a sample system.

Example 1–1. Sample Output From make -s Command

```
[SOPC Builder]$ 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 rebuild your project after significant hardware changes, you should build your project from the Nios II IDE. The Nios II IDE recreates the makefile for the new version of your system after the modifications.

Software Build Tools Flow

The Nios II software build tools flow uses the software build tools to provide a flexible, portable, and scriptable software build environment. Altera recommends that you use this flow if you prefer a command-line environment, or if you want a set of build tools that fits easily in your preferred software or system development environment. The Nios II software build tools are the basis for Altera's future development.

The software build tools flow requires that you have an SOPC file (**.sopc**) generated by SOPC Builder 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 design examples based on this flow that are shipped with every release of the Nios II EDS. For more information about these examples, refer to the "Using Nios II Example Design Scripts" section of the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Board Design Considerations

You must choose the method to configure, or program, your FPGA, and the method to boot your Nios II processor.

Configuration

Many FPGA configuration options are available to you. The two most commonly used options configure the FPGA from flash memory. One option uses a CPLD and a CFI flash device to configure the FPGA, and the other uses a serial flash EPCS configuration device. The Nios II development kits use these two configuration options by default.

Choose the first option, which uses a CPLD and a CFI-compliant flash memory, in the following cases:

- Your FPGA is large
- You must configure multiple FPGAs
- You require a large amount of flash memory for software storage
- Your design requires multiple FPGA hardware images (safe factory images and user images) or multiple software images

EPCS configuration devices are often used to configure small, single-FPGA systems.

The default Nios II boot loader does not support multiple FPGA images in EPCS devices.

For help in configuring your particular device, refer to the device family information at www.altera.com/products/devices/dev-index.jsp.

Figure 1–3 shows the block diagram of the configuration controller used on the Nios II Development Kit, Cyclone® II Edition. This controller design is used on many of the development kits, and is a good starting point for your design.

For more information about controller designs, refer to *AN346: Using the Nios Development Board Configuration Controller Reference Designs*.

© January 2009 Altera Corporation

Figure 1–3. Configuration Controller for Cyclone II Devices

altremote_update Megafunction-Based Configuration

Newer devices such as the Cyclone III, Stratix® II, and later devices include the built-in ALTREMOTE_UPDATE megafunction to help you configure your FPGA. For these newer devices, no additional Programmable Logic Device (PLD) is necessary for configuration control. However, older devices require a configuration controller device, as shown in Figure 1–3.

For information about the ALTREMOTE_UPDATE megafunction, refer to the *Remote Update Circuitry Megafunction User Guide (ALTREMOTE_UPDATE)*. The Application Selector example uses this megafunction in the Nios II Embedded Evaluation Kit (NEEK), Cyclone III Edition.

Booting

Many Nios II booting options are available. The following options are the most commonly used:

- Boot from CFI Flash
- Boot from EPCS
- Boot from on-chip RAM

The default boot loader that is included in the Nios II EDS supports boot from CFI flash memory and from EPCS flash memory. If you use an on-chip RAM that supports initialization, such as the M4K and M9K types of RAM, you can boot from the on-chip RAM without a boot loader.

For additional information on Nios II boot methodologies, refer to *AN458: Alternative Nios II Boot Methods*.

Additional Design Considerations

Consider the following topics as you design your system:

- JTAG signal integrity
- Extra memory space for prototyping
- System verification

JTAG Signal Integrity

The JTAG signal integrity on your system is very important. You must debug your hardware and software, and program your FPGA, through the JTAG interface. Poor signal integrity on the JTAG interface can prevent you from debugging over the JTAG connection, or cause inconsistent debugger behavior.

You can use the System Console to verify the JTAG chain.

JTAG signal integrity problems are extremely difficult to diagnose. To increase the probability of avoiding these problems, and to help you diagnose them should they arise, Altera recommends that you follow the guidelines outlined in *AN428: MAX II CPLD Design Guidelines* and in the *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook* when designing your board.

For more information about the System Console, refer to the *System Console User Guide*.

Extra Memory Space For Prototyping

Even if your final product includes no off-chip memory, Altera recommends that your prototype board include a connection to some region of off-chip memory. This component in your system provides additional memory capacity that enables you to focus on refining code functionality without worrying about code size. Later in the design process, you can substitute a smaller memory device to store your software.

System Verification

For useful information about design techniques for your embedded system, refer to the *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook*. Altera recommends that you read this chapter before you begin your design.

Resources

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

Support

Altera recommends that you seek support in the following order:

1. Search www.altera.com for answers to your questions.

   Relevant literature appears on the Altera literature pages, especially on the Nios II Processor literature page and the SOPC Builder literature page.

2. Contact your local Altera sales office or sales representative, or your field application engineer (FAE).
3. Contact technical support at www.altera.com/mysupport to get support directly from Altera.
4. Consult the community-owned Nios Forum and Wiki:
   - www.niosforum.com
   - www.nioswiki.com

Altera is not responsible for the contents of the Nios Forum and Nios Wiki websites, which are maintained by groups outside of Altera.

To learn how the tools work together and to use them in an instructor-led environment, register for training.

Training

Several training options are available. For information about general training, refer to Altera's Education and Events website at www.altera.com/education/edu-index.html.

For detailed information on available courses and their locations, visit the Altera Technical Training website at www.altera.com/education/training/curriculum/embedded_sw/trn-embedded_sw.html. This website contains information on both online and instructor-led training.

Documentation

Documentation about the Nios II processor and embedded design is located in your Nios II EDS installation directory at **$SOPC_KIT_NIOS2/documents/index.htm**. To access this page directly on Windows platforms, on the Start menu, click **All Programs**. On the All Programs menu, on the Altera submenu, on the Nios II EDS <*version*> submenu, click **Nios II <*version*> Documentation.** This web page contains links to the latest Nios II documentation.

The Nios II literature page includes a list and links to available documentation at www.altera.com/literature/lit-nio2.jsp. 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.

Useful information for first time Nios II IDE users appears on the Welcome page. This page appears the first time you open the Nios II IDE after a new installation. You can also open it at any time from the Nios II IDE by clicking **Welcome** on the Help menu. A PDF version is also available at www.altera.com/literature/ug/ug_nios2_ide_help.pdf.

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.

Third Party Intellectual Property

Many third parties have participated in developing solutions for embedded designs with Altera FPGAs through the Altera AMPP℠ Program. For up-to-date information on the third-party solutions available for the Nios II processor, refer to the Altera embedded processing web pages at www.altera.com/embedded, and click **Embedded Software Partners**.

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

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

Glossary

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

- **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.
- **Component—**A named module in SOPC Builder that contains the hardware and software necessary to access a corresponding hardware peripheral.
- **Custom instruction—**Custom hardware processing integrated into the Nios II processor's ALU. The programmable nature of the Nios II processor and SOPC Builder-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 off-load data transfer operations from the processor in data streaming applications.
- **ELF (Executable and Loadable 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 C-To-Hardware Acceleration (C2H) Compiler**—A push-button ANSI C-to-hardware compiler that allows you to explore algorithm acceleration and design-space options in your embedded system.
- **Nios II Command Shell**—The command shell you use to access Nios II and SOPC Builder 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 `$SOPC_KIT_NIOS2/sdk_shell` ↵
- **Nios II Embedded Development Suite (EDS)**—The complete software environment required to build and debug software applications for the Nios II processor. The EDS includes the Nios II IDE. (*See* **Nios II IDE**).
- **Nios II IDE**—An Eclipse-based development environment for Nios II embedded designs that provides software project management, build, and debugging capabilities.
- **SOPC Builder**—Software that provides a GUI-based system builder and related build tools for the creation of FPGA-based subsystems, with or without a processor.

Conclusion

This chapter is a basic overview of the Altera embedded development process and tools for the first time user. The chapter focuses on using these tools and where to find more information. It references other Altera documents that provide detailed information on the individual tools and procedures. It contains resource and glossary sections to help orient the first time user of Altera's embedded development tools for hardware and software development.

Referenced Documents

This chapter references the following documents:

- *AN346: Using the Nios Development Board Configuration Controller Reference Designs*
- *AN428: MAX II CPLD Design Guidelines*
- *AN458: Alternative Nios II Boot Methods*
- *Archiving SOPC Builder Projects* chapter in volume 4 of the *Quartus II Handbook*
- *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*
- *Developing Components for SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*
- *Developing Nios II Software* chapter of the *Embedded Design Handbook*
- *Embedded Design Handbook*

- *Hardware Acceleration and Coprocessing* chapter of the *Embedded Design Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Nios II Command-Line Tools* chapter of the *Embedded Design Handbook*
- *Nios II Custom Instruction User Guide*
- *Nios II Flash Programmer User Guide*
- *Nios II Hardware Development Tutorial*
- *Nios II Processor Reference Handbook*
- *Nios II Software Developer's Handbook*
- *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*
- *Remote Update Circuitry Megafunction User Guide (ALTREMOTE_UPDATE)*
- *System Console User Guide*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook*
- *Volume 4: SOPC Builder* of the *Quartus II Handbook*
- *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*

Document Revision History

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

Table 1–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
January 2009 v2.1	Updated Nios Wiki hyperlink.	Updated Nios Wiki hyperlink.
November 2008 v2.0	Added System Console.	Added System Console.
March 2008 v1.0	Initial release.	—

Section II. Nios II Software Development

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

This section includes the following chapters:

- Chapter 2, Developing Nios II Software
- Chapter 3, Debugging Nios II Designs
- Chapter 4, Nios II Command-Line Tools
- Chapter 5, Optimizing Nios II C2H Compiler Results

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

2. Developing Nios II Software

ED51002-1.1

Introduction

This chapter provides in-depth information about software development for the Altera® Nios® II processor. It complements the *Nios II 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 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 document, you should be familiar with the process of creating a board-support package (BSP) project and an application project using the Nios II software development flow. The new Nios II software development flow, first supported by the Nios II Embedded Design Suite (EDS) v7.1, is very different from the older Nios II Integrated Development Environment (IDE) software development flow. The following resources provide training on the new Nios II software development flow, called the Nios II software build tools flow:

- Online training demonstrations located at www.altera.com/education/training/curriculum/embedded_sw/trn-embedded_sw.html:
  - Developing Software for the Nios II Processor: Tools Overview
  - Developing Software for the Nios II Processor: Design Flow
  - Developing Software for the Nios II Processor: Software Build Flow (Part 1)
  - Developing Software for the Nios II Processor: Software Build Flow (Part 2)
- Documentation located at www.altera.com/literature/lit-nio2.jsp, especially the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- Example designs provided with the Nios II 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 chapter is structured according to the Nios II software development process. Each section describes Altera's recommended design practices to accomplish a specific task.

This chapter contains the following sections:

- "Software Development Cycle"
- "Software Project Mechanics" on page 2–5
- "Developing With the Hardware Application Layer" on page 2–16
- "Optimizing the Application" on page 2–34

- "Linking Applications" on page 2–40
- "Application Boot Loading and Programming System Memory" on page 2–42

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 chapter focuses on the use of the Altera-created tools for Nios II software generation. It also includes some discussion of third-party tools.

The Nios II EDS is a collection of software generation, management, and deployment tools for the Nios II processor. The toolchain includes tools that perform low-level tasks and tools that perform higher-level tasks using the lower-level tools.

This section contains the following subsections:

- "Altera System on a Programmable Chip (SOPC) Solutions"
- "Nios II Software Development Process" on page 2–3

Altera System on a Programmable Chip (SOPC) Solutions

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

An SOPC Builder 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 SOPC Builder system.

In Altera SOPC Builder solutions, the hardware design is implemented in an FPGA device. An FPGA device, in contrast to a normal ASIC device, is volatile—contents are lost when the power is turned off—and reprogrammable. When an FPGA is programmed, the logic cells inside it are configured and connected to create an SOPC 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 SOPC Builder—basic properties:
  - **Volatility—**The FPGA is functional only after it is configured, and it can be reconfigured at any time.
  - **Design—**Most Altera SOPC systems are designed using SOPC Builder and the Quartus® II software, and may include multiple peripherals and processors.
  - **Configuration—**FPGA configuration can be performed through a programming cable, such as the USB-Blaster™ cable, which is also used for Nios II software debugging operations.
  - **Peripherals—**Peripherals are created from FPGA resources and can appear anywhere in the Avalon memory space. Most of these peripherals are internally parameterizeable.
- 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.
  - **Parametrization**—Many properties of the Nios II processor are parameterizeable in SOPC Builder, including core type, cache memory support, and custom instructions, among others.
  - **Processor Memory—**The Nios II processor must boot from and run code loaded in an internal or external memory device.
  - **Debug support—**To enable software debug support, you must configure the Nios II processor with a debug core. Debug communication is performed through a programming cable, such as the USB-Blaster cable.
  - **Reset vector—**The reset vector address can be configured to any memory location.
  - **Exception vector—**The exception vector address can be configured to any memory location.

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

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

You create the hardware for the system using the Quartus II and SOPC Builder 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 SOPC Builder system file (**.sopc**), which is the specification file that describes the hardware components and connections.

The software generation tools use the **.sopc** 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 compiled with your application project to create an executable binary file for your system, called an application image. The combination of the BSP project and your application project is called the software project.

The application project is your application C source and header files and a makefile that you can generate by running Altera-provided tools. You can edit these files and compile 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 hardware abstraction layer (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 also import the generated project file to the Nios II IDE framework, which provides you with editing, compilation, and debugging support.

☞ In the Nios II IDE design flow, the BSP library file is called a system library.

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. Changes to the hardware require that you create a new BSP library file with updated header files. You can also modify the BSP library file to include additional Altera-supplied components, such as the read-only file system (ZIPFS) 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.

The key file required to generate the application software is the SOPC database file, the **.sopc** file. This file describes the target system hardware configuration.

Software Project Mechanics

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

This section contains the following subsections:

- "Software Tools Background"
- "Development Flow Guidelines" on page 2–6
- "Nios II Software Build Tools Flow" on page 2–6
- "Configuring BSP and Application Projects" on page 2–7
- "Software Project Development Mechanics" on page 2–10
- "Ensuring Software Project Coherency" on page 2–12

Software Tools Background

The Nios II EDS provides a sophisticated set of software project generation tools to build your application image. In version 7.2 of the Nios II EDS, two separate software-development methodologies are available for project creation—the Nios II IDE flow and the Nios II software build tools flow.

Of the two software-generation flows available to you, the Nios II IDE software-development flow predates the other. The Nios II IDE software-development flow was initially released with version 1.0 of the Nios II processor. Its goal was to provide users with a GUI environment for configuring, building, and debugging software projects. The Nios II software build tools flow was initially released in version 7.1 of the Nios II EDS. It was designed to provide users with a command-line and script-driven, easily controllable development environment for creating, managing, and configuring software applications. The Nios II IDE is still available for editing, building, and debugging software applications.

Altera recommends that you use the Nios II software build tools flow for generating new software projects. The Nios II software build tools are the basis for Altera's future development.

For information about migrating existing Nios II IDE projects to the Nios II software build tools flow, refer to the "Porting Nios II IDE Projects" section of the *Using the Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Development Flow Guidelines

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

- **Begin with a known hardware design**—The Nios II EDS includes a set of known working designs, called hardware example designs, which are excellent starting points for your own design.
- **Begin with a known software example design**—The Nios II EDS includes a set of preconfigured application and BSP 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 files include inline comments that provide additional information.
- **Make incremental changes**—Regardless of your end-application goals, develop your software application by making incremental, testable changes, to compartmentalize your software development process. Altera recommends that you use a version control system to maintain distinct versions of your source files as you develop your project.

The following section describes how to implement these guidelines.

Nios II Software Build Tools Flow

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 into an application image. The BSP project is a parameterizeable 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.

The BSP and application projects are built using the following command-line tools:

- **nios2-bsp**—This script creates a makefile that builds a BSP library file from the BSP project.
- **nios2-app-generate-makefile**—This utility creates a makefile that builds an application image from the application project and the BSP library file.

Both of these commands allow parameterization of their respective projects through the use of Tcl commands and settings.

For the full list of generators, utilities, and scripts in the Nios II software build tools flow, refer to the "Generators, Utilities, and Scripts" section of the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

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 the "Common BSP Tasks" section of the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Software Example Designs

While you are still becoming acquainted with the Nios II software build flow, the easiest way to 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. They are shell scripts that use the **nios2-bsp** and **nios2-app-generate-makefile** commands with different parameters.

For more information about the software example designs provided in the Nios II EDS, refer to the "Using Nios II Example Design Scripts" section of the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

To use a software example design, perform the following steps:

1. Set up a working directory that contains your system hardware, including the system **.sopc** file.
2. In your Quartus II installation, in the hardware project directory of your Altera Nios development board type—for example, **C:\altera\72\nios2eds\examples\verilog\niosII_cycloneII_2c35** —in **software_examples**, select an example you are interested in using.
3. Copy the entire **software_examples** directory to your working directory.
4. In the Nios II command shell, change to your chosen example directory in the new working subdirectory.
5. Type the following command at the command prompt:
   ```
   ./create-this-app ↵
   ```
   You have generated the software application image of both the application and BSP projects for your system hardware.

You must ensure that your system hardware satisfies the requirements for the software example design. If you use a standard Altera development kit, the supplied software example designs are guaranteed to work with the particular hardware configuration for that board.

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 project 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. Altera recommends that you not edit the generated files directly, because they can be overwritten by the BSP generation tools.

To configure a BSP project, Altera recommends that you create a Tcl configuration file and pass it to the **nios2-bsp** command using the `--script` option.

Selecting Core Services (HAL versus MicroC/OS-II RTOS)

You have a choice of two separate run-time environments that you can incorporate in your BSP library file. These two environments are the Nios II hardware abstraction layer (HAL) and the MicroC/OS-II real-time operating system (RTOS), which you specify as `ucosii`. The HAL environment is a lightweight, POSIX-like, single-threaded library, and is sufficient for many applications. The MicroC/OS-II RTOS enables multi-threaded processing and HAL-level services. To enable one of these two services, type the following command:

`nios2-bsp <hal` *or* `ucosii> <`*bsp-dir*`>` ↵

MicroC/OS-II RTOS Configuration Tips

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

- **UCOSII BSP settings—**The MicroC/OS-II RTOS component supports many configuration options. Some of these options are enabled by default, while others are enabled with BSP settings. A comprehensive list of options appears in the "Settings" section of the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.
- **UCOSII setting modification—**Setting or clearing the `UCOSII` options modifies the **system.h** file, which is used to compile the BSP library file.
- **UCOSII initialization—**The core MicroC/OS-II component 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" on page 2–18.
- **UCOSII configuration script—**Altera recommends that you create a configuration script to store your `UCOSII` configuration settings (Example 2–1).

Example 2–1. UCOSII 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
```

The UCOSII configuration script in Example 2–1 enables the UCOSII timer and queue code, and defines a maximum of four timers for use. To run this script during BSP generation, type the following command line:

```
nios2-bsp UCOSII . ../system.sopc --script ucosii_conf.tcl ↵
```

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 HAL configuration options appears in the "Settings" section of the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.
- **HAL setting modification**—Setting or clearing the HAL options modifies the **system.h** file, which is used to compile the BSP library file.
- **HAL initialization**—The core HAL component 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 "crt0 Initialization" on page 2–18.
- **HAL configuration script**—Altera recommends that you create a configuration script to store your HAL configuration settings (Example 2–2).

Example 2–2. HAL Tcl Configuration Script Example (hal_conf.tcl)

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

Adding Additional Components

Altera supplies several add-on software packages in the Nios II EDS. These add-on components are available for your application to use. The following components are provided:

- **Host File System**—Allows a Nios II system to access a file system that resides on the workstation. For more information, refer to "HOSTFS: Workstation-Based File System" on page 2–29.
- **Read-Only Zip File System**—Provides access to a simple file system stored in flash memory. For more information, refer to "ZIPFS: Read-Only File System" on page 2–29.
- **NicheStack TCP/IP Stack – Nios II Edition**—Enables support of the NicheStack TCP/IP networking stack component.

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 Software Developer's Handbook*.

Configuring the Application Project

Configure the application project by specifying user source files and a valid BSP project, along with other command-line options.

Application Configuration Tips

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

- **Makefile modification**—For quick experimentation, edit the generated makefile. This method is faster than regenerating the entire application project.
- **Source file inclusion**—Several options are available for specifying the user source files in your application project. If all your source files are in the same directory, use the `--src-dir` command-line option.
- **Makefile variables**—Set makefile variables with the `--set` *<var>* *<value>* command-line option during configuration of the application project. Examine a generated application makefile to ensure you understand the current and default settings.
- **Creating top level generation script**—Simplify the parameterization of your application project by creating a top level shell script to control the configuration. The **create-this-app** scripts mentioned in "Software Example Designs" on page 2–7 are good models for your configuration script.

Linking User Libraries

You can also create and use your own user libraries in the Nios II software development flow, 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.

Software Project Development Mechanics

This section describes the recommended ways to edit, build, download, run, and debug your software application, with and without the Nios II IDE.

The Role of the Nios II IDE

Although the Nios II software build tools flow is recommended for configuring your software application, the Nios II IDE is a good graphical tool for editing, debugging, and running the application on the target system. Before you can use the Nios II IDE for developing your software application, you must import the project, which includes both the application and BSP projects.

For more information, refer to the "Importing User-Managed Projects" section of the *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*.

Editing the Project

In the Nios II IDE, you can edit the application source files, BSP project files, and user library files, all of which appear in the project navigator. However, any modifications to the BSP project files are overwritten when you regenerate the BSP project.

Altera recommends that you not edit the BSP project files unless absolutely necessary, because of the project maintenance implications. Modifying the source code for the Altera-supplied BSP libraries, device drivers, or add-on components creates your own custom version of the Altera libraries. Before you edit the BSP project files, confirm that you cannot make your desired modifications with BSP settings or by modifying driver or package settings.

Building the Project

To build your application, use the makefiles created for the application and BSP projects. These makefiles use the Nios II GNU toolchain, which is provided with the Nios II EDS.

Alternatively, you can use the TASKING VX toolset to build your application. This toolset is available for purchase from Altium Limited (www.altium.com).

Downloading and Running the Software

From the command line, download and run your application image by typing the following command:

```
nios2-download -g <myapp>.elf ↵
```

This command line downloads the application image **.elf** file to the target device and runs the **.elf** file.

Before you run your target application, ensure that your FPGA is configured with your target hardware image.

In the Nios II IDE environment, you must import the BSP and application projects and make a **Run** or **Debug** configuration for your project.

For information about using the Nios II IDE to download and run the software application, refer to the *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*.

Communicating with the Target

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. Unfortunately, the Nios II IDE 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.

The Nios II IDE automatically provides a console window in which you can communicate with your system. When you use the Nios II IDE to communicate with the target, characters you input are transmitted to the target line by line. Characters are visible to the target only after the Enter key is pressed on your keyboard.

Software Debugging

The Nios II IDE helps you to debug the application by providing breakpoint, source navigation, and memory viewing support. To use the Nios II IDE in debug mode, you must create and run a debug configuration, which downloads the **.elf** file and runs the debugger.

Alternatively, you can debug your application using the Tcl/Tk-based Insight GDB GUI, which installs with the Nios II EDS distribution, or using a third party debugger.

For more information about using the Nios II IDE to debug your application, refer to the *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*.

Enabling the `hal.enable_runtime_stack_checking` setting when you configure your BSP project turns on stack checking. This setting causes subroutine calls to generate an exception if the stack collides with the heap or with statically allocated data in memory.

For more information about this and other BSP configuration settings, refer to the "Settings" section of the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Ensuring Software Project Coherency

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

Recommended Development Practice

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

One best practice is to structure your application hierarchy with parallel application project and BSP project folders, as in the Nios II installation **software_examples** directories. In Figure 2–1, a top-level hardware project folder includes the Quartus II project file, the SOPC Builder-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 2–1. Recommended Directory Structure

- hardware project folder
  - <*system_name*>.sopc
  - <*system_name*>.ptf
  - <*system_name*>.sof
  - software project folder
    - APP project folder
      - create-this-app
      - clean-this-app
      - application software source files
    - BSP project folder
      - create-this-bsp
      - clean-this-bsp
      - bsp_settings.tcl (optional) (1)

Note for Figure 2–1:

(1) is a Tcl configuration file. For more information about the Tcl configuration file, refer to "Configuring the BSP Project" on page 2–7.

For your own software project, you must create the **create-this-app** and **create-this-bsp** scripts. Altera recommends that you also create **clean-this-app** and **clean-this-bsp** scripts. These scripts perform the following tasks:

- **create-this-app**—This **bash** script uses the **nios2-app-generate-makefile** command to create the application project, using the application software source files for your project. The script verifies that the BSP project was properly configured (a **settings.bsp** file is present in the BSP project directory), and runs the **create-this-bsp** script if necessary. The Altera-supplied **create-this-app** scripts that are included in the software project example designs 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***>.sopc** file located in the hardware project folder. Running this script creates all the BSP project files 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 SOPC Builder. The system generation process follows:

1. **Hardware files generation**—Using SOPC Builder, write the updated system description to the **<***system_name***>.sopc** and **<***system_name***>.ptf** files.
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 also runs the makefile to generate the BSP library file.
4. **Build the system**—Build the system software using the application and BSP makefile scripts.

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

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**—At minimum, the hardware group provides copies of the <*system_name*>**.sopc**, <*system_name*>**.ptf**, 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 user 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.

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 SOPC Builder, 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, address location, and address span. To protect against coherency issues, access all system peripherals and memory components with their **system.h** name and address span symbols. This method guarantees access regardless of a peripheral's addressable location.

  For example, if your system includes a UART peripheral named UART1, located at address 0x1000, access it 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 through the System ID core**—Use the System ID core. The System ID core is an SOPC Builder peripheral that provides a unique identifier for a generated system. This identifier is stored in a hardware register readable by the Nios II processor. This unique identifier is also stored in the **.sopc** 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 two methods. The first method is automatically implemented during system software development, when the **.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. The second method for using the system ID peripheral is useful in systems that do not have a Nios II debug port, or in situations in which running the Nios II software download utilities is not practical. In this method you use the C function `alt_avalon_sysid_test()`. This function reports whether the hardware and software system IDs match.

For more information about the System ID core, refer to the *System ID Core* chapter in volume 5 of the *Quartus II Handbook*.

Developing With the Hardware Application Layer

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

This section contains the following subsections:

■ "Overview of the HAL" on page 2–16
■ "System Startup in HAL-Based Applications" on page 2–17
■ "HAL Peripheral Services" on page 2–20
■ "Accessing Memory With the Nios II Processor" on page 2–31
■ "Handling Exceptions" on page 2–33
■ "Modifying the Exception Handler" on page 2–34

Overview of the HAL

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

HAL Configuration Options

To support the Nios II software development flow, the HAL system 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 set 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" on page 2–7, Altera recommends you create a separate Tcl file that contains your HAL configuration settings.

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

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 non-volatile memory for execution, a boot loader must configure the application image in the non-volatile memory. This configuration process occurs before the HAL system library configuration routines execute, and before the crt0 code block executes. A boot loader implements this process. For more information, refer to "Linking Applications" on page 2–40 and "Application Boot Loading and Programming System Memory" on page 2–42.

Controlling HAL Initialization

As noted in "HAL Initialization" on page 2–19, although most user applications begin execution in a main() function, some applications 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.

Minimizing the Code Footprint and Increasing Performance

For information about increasing your application's performance, or minimizing the code footprint, refer to "Optimizing the Application" on page 2–34.

Configuring Peripherals and Services

For information about configuring and using HAL services, refer to "HAL Peripheral Services" on page 2–20.

System Startup in HAL-Based Applications

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

System Initialization

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

1. **Hardware reset event**—The board receives a power-on reset signal, which resets the FPGA.
2. **FPGA configuration**—The FPGA is programmed with a **.sof**, 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 SOPC Builder 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 pre-configured 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" on page 2–42.
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.

crt0 Initialization

The `crt0` code block contains the C run-time initialization code—software instructions needed to enable execution of C or C++ applications, and potentially usable for assembly language as well. The Altera-provided `crt0` block performs the following initialization steps:

1. **Calls alt_load macros**—If the application is designed to run from flash memory (the `.text` section runs from flash memory), the remaining sections are copied to volatile memory. For additional information, refer to "Configuring the Boot Environment" on page 2–17.
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.

   SOPC Builder determines the processors that have instruction caches, and configures these caches at system generation. The 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" on page 2–40.
5. **Clears the .bss section**—The `.bss` section is initialized. You can set the `.bss` section address. For additional information refer to "HAL Linking Behavior" on page 2–40.
6. **Initializes stack overflow protection**—Stack overflow checking is initialized. For additional information, refer to "Software Debugging" on page 2–12.
7. **Jumps to alt_main**—The processor jumps to the `alt_main` code block, which begins initializing the HAL system library.

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

© June 2008 Altera Corporation

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. These optimizations are controlled by the `hal.enable_sim_optimize` BSP setting, documented in the "Settings" section of the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

After you generate your BSP project, the **crt0.s** source file is located in the **HAL/src** directory.

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" on page 2–18. After `crt0.s` completes the C run-time initialization, it calls the HAL `alt_main()` function, which initializes the HAL system 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 real-time operation system (RTOS), if this RTOS is configured to run (with the `ALT_OS_INIT` and `ALT_SEM_CREATE` functions). For additional information on MicroC/OS-II use and initialization, refer to "Selecting Core Services (HAL versus MicroC/OS-II RTOS)" on page 2–8.
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. You can override this behavior in the BSP project by using the `--cmd set_driver` <*peripheral_name*> `none` command-line option in the call to the **nios2-bsp** script. For information about removing a device configuration function, refer to "Optimizing the Application" on page 2–34.
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" on page 2–22.
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 "Optimizing the Application" on page 2–34.
6. **Calls main()**—Calls user function `main()`, or application program. Most user applications are constructed using a `main()` function declaration, and begin execution at this function.

If you use a system library other than 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 **$SOPC_KIT_NIOS2\examples\software\hello_alt_main**.

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

HAL Peripheral Services

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

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

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

Timers

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

For more information about this peripheral, refer to the *Timer Core* chapter in volume 5 of the *Quartus II Handbook*.

System Clock Timer

The system clock timer resource is used to trigger periodic events—alarms— and as a time-keeping device that counts system clock ticks. The system clock timer service requires that a timer peripheral be present in the SOPC Builder system. This timer peripheral must be dedicated to the HAL system clock timer service.

Only one system clock timer service may be identified in the BSP library file. 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. Altera provides separate APIs for user-level system clock functionality and for generating alarms.

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

The POSIX-like `getttimeofday()` 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 SOPC Builder.
- **Rollover**—The internal, global variable that stores the number of system clock counts (since reset) is a 32-bit unsigned integer. No rollover protection is offered for this variable. Therefore, you should calculate when the rollover event will occur in your system, and plan the application accordingly.
- **Performance Impact**—Every clock tick causes the execution of an interrupt service routine. Executing this routine leads to a minor performance penalty. If your system hardware specifies a short timer period, the cumulative interrupt latency may impact your overall system performance.

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

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

- **Interrupt Service Routine (ISR) context**—A common mistake is to program the alarm callback function to call a service that depends on interrupts being enabled (such as the `printf()` function). This mistake causes the system to deadlock, because the alarm callback function occurs in an interrupt context, while interrupts are disabled.
- **Resetting the alarm**—The callback function can reset the alarm by returning a non-zero 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 SOPC Builder 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. Altera provides a timestamp API.

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

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

- **Timer Frequency**—The timestamp timer decrements at the clock rate of the clock that feeds it in the SOPC Builder system. You can modify this frequency in SOPC Builder.
- **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 $((1/\text{timer frequency}) \times 2^{32})$ seconds.

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

Character Mode Devices

stdin, stdout, and stderr

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

Currently, Altera supplies two system components that can support the `stdio` library, the UART and JTAG UART components. These devices can function as standard I/O devices. To enable this functionality, use the `--default_stdio` *<device>* option during Nios II BSP configuration.

The `stdin` character input file variable and the `stdout` and `stderr` character output file variables can also be individually configured with the HAL BSP settings `hal.stdin`, `hal.stdout`, and `hal.stderr`.

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 through the Nios II EDS development tools. For more information about performing this task, refer to "Communicating with the Target" on page 2–11.

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

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 Example 2–3 specifies that a device that is already open is to function in non-blocking mode:

Example 2–3. 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);
```

The code fragment in Example 2–4 illustrates the use of a non-blocking device:

Example 2–4. 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.

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*.

Flash Memory Devices

The HAL system library supports parallel common flash interface (CFI) memory devices and Altera erasable, programmable, configurable serial (EPCS) flash memory devices. A uniform API is available for both flash memory types, providing users with 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 were not explicitly omitted and the small driver configuration was 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 Software Developer's Handbook*.

Configuration and Use Limitations

If you use flash memories in your system, be aware of the following properties of this memory:

- **Code Storage**—If your application runs code directly from the flash memory, the flash manipulation functions are disabled. This setting prevents the processor from erasing the memory that holds the code it is running. In this case, the symbols `ALT_TEXT_DEVICE`, `ALT_RODATA_DEVICE`, and `ALT_EXCEPTIONS_DEVICE` must all have values different from the flash memory peripheral. (Note that each of these `#define` symbols names a memory device, not an address within a memory device).
- **Small Driver**—If the small driver flag is set for the software—the `hal.enable_reduced_device_drivers` setting is enabled—then the flash memory peripherals are not automatically initialized. In this case, your application must call the initialization routines explicitly.
- **Thread safety**—Most of the flash access routines are not thread-safe. If you use any of these routines, construct your application so that only one thread in the system accesses these function.
- **EPCS flash memory limitations**—The Altera EPCS memory has a serial interface. Therefore, it cannot run Nios II instructions and is not visible to the Nios II processor as a standard random-access memory device. Use the Altera-supplied flash memory access routines to read data from this device.
- **File System**—The HAL flash memory API does not support a flash file system in which data can be stored and retrieved using a conventional file handle. However, you can store your data in flash memory before you run your application, using the ZIPFS file system and the Nios II flash programmer utility. For information about the ZIPFS file system, refer to "ZIPFS: Read-Only File System" on page 2–29.

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 Software Developer's Handbook*.

For more information about the API for the flash memory access routines, refer to the *HAL API Reference* appendix of the *Nios II Software Developer's Handbook*.

Direct Memory Access (DMA) Devices

The HAL 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 the "Using DMA Devices" section of the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II 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.

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 the "Using DMA Devices" section of the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

DMA Data-Width Parameter

The DMA data-width parameter is configured in SOPC Builder 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 4 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 multiple 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" on page 2–38**.**

For more information about the HAL API for accessing DMA devices, refer to the "Using DMA Devices" section of the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook* and to the *HAL API Reference* appendix of the *Nios II Software Developer's Handbook*.

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 user access to files. In addition, the HAL provides two file systems, one that enables a Nios II system to access the workstation's file system (HOSTFS), and a simple read-only file system that enables access to the system's on-board files (ZIPFS).

Several conventional (read/ write) file systems are available through third-party vendors. For up-to-date information about the file system solutions available for the Nios II processor, refer to the Altera embedded processing web pages at www.altera.com/embedded, and click **Embedded Software Partners**.

HOSTFS: Workstation-Based File System

The HOSTFS 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 HOSTFS 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**—HOSTFS is only available during debug sessions from the Nios II IDE. Therefore, you should use HOSTFS only during system debugging and prototyping operations.
- **Incompatibility with direct drivers—**HOSTFS only works if the HAL system library is configured with direct driver mode disabled. However, enabling this mode reduces the size of the application image. For more information, refer to "Optimizing the Application" on page 2–34**.**

For more information, refer to the Nios II IDE online Help and the host file system Nios II software example design listed in the "Using Nios II Example Design Scripts" section of the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

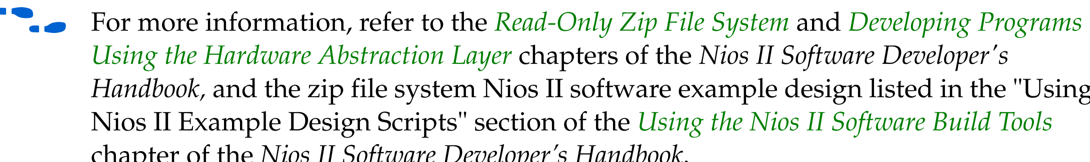
ZIPFS: Read-Only File System

ZIPFS was designed to be a lightweight, read-only file system for the Nios II processor, targeting flash memory.

Consider the following points about the ZIPFS file system before you use it:

- **Read Only—**ZIPFS is a read-only file system.

- **Configuring the file system—**To create the 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—**ZIPFS only works if the HAL system library is configured with direct driver mode disabled. However, enabling this mode reduces the size of the application image. For more information, refer to "Optimizing the Application" on page 2–34.


For more information, refer to the *Read-Only Zip File System* and *Developing Programs Using the Hardware Abstraction Layer* chapters of the *Nios II Software Developer's Handbook*, and the zip file system Nios II software example design listed in the "Using Nios II Example Design Scripts" section of the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Ethernet Devices

Ethernet devices are a special case for the HAL service model. To make them accessible to the application, these devices require an additional software library, a TCP/IP stack. Altera supplies a TCP/IP networking stack called NicheStack, which provides your application with a socket-based interface for communicating over Ethernet networks.

For more information, refer to the *Ethernet and the NicheStack TCP/IP Stack – Nios II Edition* chapter of the *Nios II Software Developer's handbook*.

Unsupported Devices

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

- Altera's third-party program supports your device
- You can incorporate your own device

Altera's third party program information is available on the Nios II embedded software partners page. Refer to the Altera embedded processing web pages at www.altera.com/embedded, and click **Embedded Software Partners**.

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 Software Developer's Handbook*.

Accessing Memory With the Nios II Processor

It can be difficult to create software applications that program the Nios II processor to interact correctly with data and instruction caches when it reads and writes to peripherals and memories. There are also subtle differences in how the different Nios II processor cores handle these operations, that can cause problems when you migrate from one Nios II processor core to another.

This section helps you avoid the most common pitfalls. It provides background critical to understanding how the Nios II processor reads and writes peripherals and memories, and describes the set of software utilities available to you, as well as providing sets of instructions to help you avoid some of the more common problems in programming these read and write operations.

Creating General C/C++ Applications

You can write most C/C++ applications without worrying about whether the processor's read and write operations bypass the data cache. However, you do need to make sure the operations do not bypass the data cache in the following cases:

- Your application must guarantee that a read or write transaction actually reaches a peripheral or memory.
- Your application shares a block of memory with another processor or Avalon interface master peripheral.

Accessing Peripherals

If your application accesses peripheral registers, or performs only a small set of memory accesses, Altera recommends that you use the default HAL I/O macros, IORD and IOWR. These macros guarantee that the accesses bypass the data cache.

Two types of cache-bypass macros are available. The HAL access routines whose names end in _32DIRECT, _16 DIRECT, and _8 DIRECT interpret the offset as a byte address. The other routines treat this offset as a count to be multiplied by four bytes, the number of bytes in the 32-bit connection between the Nios II processor and the system interconnect fabric. The _32DIRECT, _16DIRECT, and _8DIRECT routines are designed to access memory regions, and the other routines are designed to access peripheral registers.

Example 2–5 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 2–5. 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);
}
```

Example 2–6 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 2–6. 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);
```

Altera recommends that you use the HAL-supplied macros for accessing external peripherals and memory.

Sharing Uncached Memory

If your application must allocate some memory, operate on that memory, and then share the memory region with another peripheral (or processor), use the HAL-supplied `alt_uncached_malloc()` and `alt_uncached_free()` functions. Both of these functions operate on pointers to bypass cached memory.

To share uncached memory between a Nios II processor and a peripheral, perform the following steps:

1. **malloc memory**—Run the `alt_uncached_malloc()` function to claim a block of memory from the heap. If this operation is successful, the function returns a pointer that bypasses the data cache.
2. **Operate on memory**—Have the Nios II processor read or write the memory using the pointer. Your application can perform normal pointer-arithmetic operations on this pointer.
3. **Convert pointer**—Run the `alt_remap_cached()` function to convert the pointer to a memory address that is understood by external peripherals.
4. **Pass pointer**—Pass the converted pointer to the external peripheral to enable it to perform operations on the memory region.

Sharing Memory With Cache Performance Benefits

Another way to share memory between a data-cache enabled Nios II processor and other external peripherals safely without sacrificing processor performance is the delayed data-cache flush method. In this method, the Nios II processor performs operations on memory using standard C or C++ operations until it needs to share this memory with an external peripheral.

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 perform the following 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.

Example 2–7 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 2–7. 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;
```

Example 2–8 shows an implementation of delayed data-cache flushing for memory accesses to a memory region the Nios II processor acquired with malloc.

Example 2–8. Data-Cache Flushing With Memory Acquired Using malloc

```
char * data = (char*)malloc(sizeof(char) * 1000);

write_operands(data);
alt_dcache_flush(data, sizeof(char) * 1000);
signal_master(data);
result = read_results(data);
free(data);
```

The `alt_dcache_flush_all()` function call flushes the entire data cache, but this function is not efficient. Altera recommends that you flush from the cache only the entries for the memory region that you make available to the other master peripheral.

Handling Exceptions

The HAL infrastructure provides users with 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.

For information about the exception handler software routines, HAL-provided services, and programmer API, refer to the *Exception Handling* chapter of the *Nios II 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 IRQ prioritization of your peripherals in SOPC Builder.
- **Nesting of interrupts**—The HAL infrastructure allows interrupts to be nested—higher priority interrupts can preempt processor control from an exception handler that is servicing a lower priority interrupt. However, Altera recommends that you not nest your interrupts because of the associated performance penalty.
- **Exception handler environment**—When creating your exception handler, you must ensure that the handler does not run interrupt-dependent functions and services, because this can cause deadlock. For example, an exception handler should not call the IRQ-driven version of the `printf()` function.

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 SOPC Builder.
- **Modifying the exception handler**—The HAL-provided exception handler is fairly robust, reliable, and efficient. Modifying the exception handler could break the HAL-supplied user 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" on page 2–38.

Optimizing the Application

This section examines techniques to increase your software application's performance and decrease its size.

This section contains the following subsections:

- "Performance Tuning Background"
- "Speeding Up System Processing Tasks" on page 2–35
- "Accelerating Interrupt Service Routines" on page 2–38
- "Reducing Code Size" on page 2–38

Performance Tuning Background

Software performance is the speed with which a certain task or series of tasks can be performed in the system. To increase software performance, you must first determine the sections of the code in which the processing time is spent.

An application's tasks can be divided into interrupt tasks and system processing tasks. Interrupt task performance is the speed with which the processor completes an interrupt service routine to handle an external event or condition. System processing task performance is the speed with which the system performs a task explicitly described in the application code.

A complete analysis of application performance examines the performance of the system processing tasks and the interrupt tasks, as well as the footprint of the software image.

Speeding Up System Processing Tasks

To increase your application's performance, determine how you can speed up the system processing tasks it performs. First analyze the current performance and identify the slowest tasks in your system, then determine whether you can accelerate any part of your application by increasing processor efficiency, creating a hardware accelerator, or improving the applications's methods for data movement.

Analyzing the Problem

The first step to accelerate your system processing is to identify the slowest task in your system. Altera provides the following tools to profile your application:

- **GNU Profiler—**The Nios II EDS toolchain includes a method for profiling your application with the GNU Profiler. This method of profiling reports how long various functions run in your application.
- **High resolution timer—**The interval timer peripheral is a simple time counter that can determine the amount of time a given subroutine runs.
- **Performance counter peripheral—**The performance counter unit can profile several different sections of code with a collection of counters. This peripheral includes a simple software API that enables you to print out the results of these counters through the Nios II processor's `stdio` services.

Use one or more of these tools to determine the tasks in which your application is spending most of its processing time.

For more information about how to profile your software application, refer to *AN391: Profiling Nios II Systems*.

Accelerating your Application

This section describes several techniques to accelerate your application. Because of the flexible nature of the FPGA, most of these techniques modify the system hardware to improve the processor's execution performance. This section describes the following performance enhancement methods:

- Methods to increase processor efficiency
- Methods to accelerate select software algorithms using hardware accelerators

- Using a DMA peripheral to increase the efficiency of sequential data movement operations

Increasing Processor Efficiency

An easy way to increase the software application's performance is to increase the rate at which the Nios II processor fetches and processes instructions, while decreasing the number of instructions the application requires. The following techniques can increase processor efficiency in running your application:

- **Processor clock frequency**—Modify the processor clock frequency using SOPC Builder. 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 SOPC Builder:
  - **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, Nios II/s, 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 in SOPC Builder* chapter of the *Nios II 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. 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 Nios II Tightly Coupled Memory Tutorial* and to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

- **Compiler Settings**—More efficient code execution can be attained through the use of compiler optimizations. Increase the compiler optimization setting to `-03`, the fastest compiler optimization setting, to attain more efficient code execution.

For information about configuring the compiler optimization level, refer to the `hal.make.bsp_cflags_optimization` BSP setting in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Accelerating Hardware

Slow software algorithms can be accelerated through the use of custom instructions, dedicated hardware accelerators, and use of the C-to-Hardware (C2H) compiler tool. The following techniques can increase processor efficiency in running your application:

- **Custom instructions**—Use custom instructions to augment the Nios II processor's 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 the *Using Nios II Floating-Point Custom Instructions* tutorial.

- **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 of the *Embedded Design Handbook*.

- **C2H Compiler**—Use the C2H Compiler to accelerate standard ANSI C functions by converting them to dedicated hardware blocks.

  For more information about the C2H Compiler, refer to the *Nios II C2H Compiler User Guide* and to the *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*.

Improving Data Movement

If your application performs many sequential data movement operations, a DMA peripheral might increase the efficiency of these operations. Altera provides the following two DMA peripherals for your use:

- **DMA**—Simple DMA peripheral that can perform single operations before being serviced by the CPU. For more information about using the DMA peripheral, refer to "HAL Peripheral Services" on page 2–20.

  For information about the DMA peripheral, refer to the *DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

- **Scatter-Gather DMA (SGDMA)**—Descriptor-based DMA peripheral that can perform multiple operations before being serviced by CPU.

  For more information, refer to the *Scatter-Gather DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

Accelerating Interrupt Service Routines

To increase the efficiency of your interrupt service routines, determine how you can speed up the tasks they perform. First analyze the current performance and identify the slowest parts of your interrupt dispatch and handler time, then determine whether you can accelerate any part of your interrupt handling.

Analyzing the Problem

The total amount of time consumed by an interrupt service routine is equal to the latency of the HAL interrupt dispatcher plus the interrupt handler running time. Use the following methods to profile your interrupt handling:

- **Interrupt dispatch time**—Calculate the interrupt handler entry time using the method found in design files that accompany the *Using Nios II Tightly Coupled Memory Tutorial* on the Altera literature pages. You can download the design files from the Nios II literature web page at www.altera.com/literature/lit-nio2.jsp.
- **Interrupt service routine time**—Use a timer to measure the time from the entry to the exit point of the service routine.

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.
- **IRQ priority**—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.
- **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.

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 Software Developer's Handbook*.

Reducing Code Size

Reducing the memory space required by your application image also enhances performance. This section describes how to measure and decrease your code footprint.

Analyzing the Problem

The easiest way to analyze your application's code footprint is to use the GNU Binary Utilities tool **nios2-elf-size**. This tool analyzes your compiled **.elf** binary file and reports the total size of your application, as well as the subtotals for the `.text`, `.data`, and `.bss` code sections. Example 2–9 shows a **nios2-elf-size** command response.

Example 2–9. Example Use of nios2-elf-size Command

```
> nios2-elf-size -d application.elf
text data bss dec hex filename
203412 8288 4936 216636 34e3c application.elf
```

Reducing the Code Footprint

The following methods help you to reduce your code footprint:

- **Compiler options**—Setting the `-Os` flag for the GCC causes the compiler to apply size optimizations for code size reduction. Use the `hal.make.bsp_cflags_optimization` BSP setting to set this flag.
- **Reducing the HAL footprint**—Use the HAL system library configuration settings to reduce the size of the HAL system library component of your BSP library file. However, enabling the size-reduction settings for the HAL system library often impacts the flexibility and performance of the system. The configuration settings for size optimization are as follows:
  - **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\"`

For more information about these settings, refer to the "Setting"s section of the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*. For an example, refer to the BSP project **hal_reduced_footprint,** included in your Quartus II installation, in the hardware project directory of your Altera Nios development board type, in **software_examples/bsp/hal_reduced_footprint**.

- **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 system 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 ZIPFS file system, whose presence increases the overall footprint of the application. However, the HAL does not enable these modules by default.

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.

This section contains the following subsections:

- "Background"
- "Linker Sections and Application Configuration"
- "HAL Linking Behavior" on page 2–40

Background

The **create-this-bsp** script and the underlying **nios2-bsp** script are responsible for creating two linker-related files for your project, **linker.x** and **linker.h**. **linker.x** is the linker command file that the generated application's makefile uses to create the **.elf** binary file. All linker setting modifications you make to the HAL BSP project affect the contents of these two files.

Linker Sections and Application Configuration

Every Nios II application contains `.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack` sections. Additional user sections can be added to the **.elf** file to hold user code and data.

These sections are placed in named memory regions, defined to correspond with physical memory devices and addresses. By default, these sections are automatically generated by the HAL. However, you can control them for a particular application.

HAL Linking Behavior

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

Default BSP Linking

During BSP configuration, the tools perform the following steps automatically:

1. **Assign memory region names**—Assign a name to each system memory device, and add each name to the linker file as a memory region.
2. **Find largest memory**—Identify the largest read-and-write memory region in the linker file.
3. **Assign sections**—Place the default sections (`.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack`) in the memory region identified in the previous step.
4. **Write files**—Write the **linker.x** and **linker.h** files.

Usually, this section allocation scheme works during the software development process, because the application is guaranteed to function if the memory is large enough.

The rules for the HAL default linking behavior are contained in the Altera-generated Tcl scripts **bsp-set-defaults.tcl** and **bsp-linker-utils.tcl** found in the **sdk2/bin** directory. These scripts are called by the **nios2-bsp-create-settings** configuration application. Do not modify these scripts directly.

User-Controlled BSP Linking

You can control the default linking behavior of the BSP tools by calling certain Tcl functions during BSP configuration. You can incorporate these functions in a Tcl script called by the **nios2-bsp-create-settings** or **nios2-bsp** command, or pass them to one of these commands as an argument. You should not modify the Altera-generated scripts, but you can write scripts that override their behavior. The following two commands are useful for manipulating linker sections:

- `add_memory_region`—Maps a memory region name to a physical memory device
- `add_section_mapping`—Maps a section name to a memory region

For more information about the linker-related BSP configuration commands, refer to the *Nios II Software Built Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

You can override the default linking behavior of the BSP configuration tools by creating a Tcl script and passing it to the **nios2-bsp** tool as an argument. Example 2–10 shows a Tcl configuration script **mem_link.tcl** that is called with the following command:

```
nios2-bsp HAL . ../system.sopc --script mem_link.tcl ↵
```

Example 2–10. Example Tcl (mem_link.tcl) File for Configuring Memory Linking and Boot Loading

```
# The names used below are created by the BSP generation tools
# We are just assigning some variables for convenience
set text_region_name ext_flash
set data_region_name ddr_sdram_0
# Add our own memory region
add_memory_region onchip_ram0 onchip_ram0 0 0x100000
# Set up our linker sections
add_section_mapping .text $text_region_name
add_section_mapping .rodata $data_region_name
add_section_mapping .rwdata $data_region_name
add_section_mapping .bss $data_region_name
add_section_mapping .heap $data_region_name
add_section_mapping .stack $data_region_name
add_section_mapping .myown onchip_ram0
# Configure the boot loader facilities
set_setting hal.linker.allow_code_at_reset 1
set_setting hal.linker.enable_alt_load 1
set_setting hal.linker.enable_alt_load_copy_rwdata 1
set_setting hal.linker.enable_alt_load_copy_rodata 1
set_setting hal.linker.enable_alt_load_copy_exceptions 1
```

To create the script in Example 2–10, you must know the default names of the memory regions created by the HAL BSP generator. The script also uses a non-default memory region called **onchip_ram0** that you must create in SOPC Builder. The `add_section_mapping` commands locate the default sections and map your own section, called `.myown`, to your custom region called `onchip_ram0`. The `hal.linker` commands in the script are explained in "Application Boot Loading and Programming System Memory".

The **nios2-bsp** script automatically creates memory region names for all memory components discovered in system hardware. The names of these memory regions are the names assigned in SOPC Builder. After the initial **settings.bsp** file is generated, you can run the following two commands to discover the default memory regions and section mappings:

- Discover the names, base addresses, and spans of all the memory regions in your system by running the following command:

```
nios2-bsp-query-settings --settings settings.bsp --cmd puts \
[get_current_memory_regions] ↵
```

- Discover the section mappings by running the following command:

```
nios2-bsp-query-settings --settings settings.bsp --cmd puts \
[get_current_section_mappings] ↵
```

Application Boot Loading and Programming System Memory

Most Nios II systems require some method to configure the hardware and software images in system memory before the processor can begin executing your application program. This section describes various possible memory topologies for your system (both volatile and non-volatile), their use, their requirements, and their configuration. The Nios II software application requires a boot loader application to configure the system memory if the system software is stored in flash memory, but is configured to

run from volatile memory. If the Nios II processor is running from flash memory—the `.text` section is in flash memory—a copy routine, rather than a boot loader, loads the other program sections to volatile memory. In some cases, such as when your system application occupies internal FPGA memory, or is pre-loaded into external memory by another CPU, no configuration of the system memory is required.

This section contains the following subsections:

■ "Default BSP Boot Loading Configuration"
■ "Boot Configuration Options" on page 2–43
■ "Generating and Programming System Memory Images" on page 2–47

Default BSP Boot Loading Configuration

The **nios2-bsp** script determines whether the system requires a boot loader and whether to enable the copying of the default sections.

By default, the **nios2-bsp** script makes these decisions using the following rules:

- **Boot loader**—The **nios2-bsp** script assumes that a boot loader is being used if the following conditions are met:
  - The Nios II processor's reset address is not in the `.text` section.
  - The Nios II processor's reset address is in flash memory.
- **Copying default sections**—The **nios2-bsp** script enables the copying of the default volatile sections if the Nios II processor's reset address is set to an address in the `.text` section.

If the default boot loader behavior is appropriate for your system, you do not need to intervene in the boot loading process.

Boot Configuration Options

You can modify the default **nios2-bsp** script behavior for application loading by using the following settings:

- **hal.linker.allow_code_at_reset**
- **hal.linker.enable_alt_load**
- **hal.linker.enable_alt_load_copy_rwdata**
- **hal.linker.enable_alt_load_copy_exceptions**
- **hal.linker.enable_alt_load_copy_rodata**

If you enable these settings, you can override the BSP's default behavior for boot loading. Altera recommends that you list the settings in a Tcl script that you pass to the BSP generation tools. Example 2–10 on page 2–42 shows such a script.

These settings are created in the **settings.bsp** configuration file whether or not you override the default BSP generation behavior. However, you may override their default values.

For more information about BSP configuration settings, refer to the "Settings" section in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

For more information about boot loading options and for advanced boot loader examples, refer to *AN458: Alternative Nios II Boot Methods*.

Booting and Running From Flash Memory

If your program is loaded in and runs from flash memory, the application's `.text` section is not copied. However, during C run-time initialization—execution of the `crt0` code block—some of the other code sections may be copied to volatile memory in preparation for running the application.

For more information about the behavior of the `crt0` code, refer to "crt0 Initialization" on page 2–18.

Altera recommends that you avoid this configuration during the normal development cycle because downloading the compiled application requires reprogramming the flash memory. In addition, software breakpoint capabilities are not available through the debugger when using this configuration.

Prepare for BSP configuration by performing the following steps to configure your application to boot and run from flash memory:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in flash memory. Configure the reset address and flash memory addresses in SOPC Builder.
2. **Text section linker setting**—Ensure that the `.text` section maps to the flash memory address region (for example, with the command `add_section_mapping .text ext_flash`) in the Tcl settings file.
3. **Other sections linker setting**—Ensure that all of the other sections, with the possible exception of the `.rodata` section, are mapped to volatile memory regions. The `.rodata` section can map to a flash-memory region.
4. **HAL C run-time configuration settings**—Use the following HAL C run-time configuration settings:
   - **hal.linker.allow_code_at_reset** `1`
   - **hal.linker.enable_alt_load** `1`
   - **hal.linker.enable_alt_load_copy_rwdata** `1`
   - **hal.linker.enable_alt_load_copy_exceptions** `1`
   - **hal.linker.enable_alt_load_copy_rodata** `1`

If your application contains custom, user-defined memory sections, you must manually load the custom sections. Use the `alt_load_section()` HAL library function to ensure that these sections are loaded before your program runs.

The HAL system library disables the flash memory service to prevent accidental override of the application image.

Booting From Flash Memory and Running From Volatile Memory

If your application image is stored in flash memory, but executes from volatile memory with assistance from a boot loader program, prepare for BSP configuration by performing the following steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is an address in flash memory. Configure this option using SOPC Builder.
2. **Text section linker setting**—Ensure that the `.text` section maps to a volatile region of system memory, and not to the flash memory.
3. **Other sections linker setting**—Ensure that all of the other sections, with the possible exception of the `.rodata` section, are mapped to volatile memory regions. The `.rodata` section can map to a flash-memory region.
4. **HAL C run-time configuration settings**—Use the following HAL C run-time configuration settings:
   - **hal.linker.allow_code_at_reset** `0`
   - **hal.linker.enable_alt_load** `0`
   - **hal.linker.enable_alt_load_copy_rwdata** `0`
   - **hal.linker.enable_alt_load_copy_exceptions** `0`
   - **hal.linker.enable_alt_load_copy_rodata** `0`

Booting and Running From Volatile Memory

This configuration is use in cases where the Nios II processor's memory is loaded externally by another processor or interconnect switch fabric master port. In this case, prepare for BSP configuration by performing the same steps as in "Booting From Flash Memory and Running From Volatile Memory", except that the Nios II processor reset address should be changed to the memory that holds the code that the processor executes initially. Prepare for BSP configuration by performing the following steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in volatile memory. Configure this option using SOPC Builder.
2. **Text section linker setting**—Ensure that the `.text` section maps to the reset address memory.
3. **Other sections linker setting**—Ensure that all of the other sections, including the `.rodata` section, also map to the reset address memory.
4. **HAL C run-time configuration settings**—Use the following HAL C run-time configuration settings:
   - **hal.linker.allow_code_at_reset** `1`
   - **hal.linker.enable_alt_load** `0`
   - **hal.linker.enable_alt_load_copy_rwdata** `0`
   - **hal.linker.enable_alt_load_copy_exceptions** `0`
   - **hal.linker.enable_alt_load_copy_rodata** `0`

This type of boot loading and sequencing requires additional supporting hardware modifications, which are beyond the scope of this chapter.

Booting From Altera EPCS Memory and Running From Volatile Memory

This configuration is a special case of the configuration described in "Booting From Flash Memory and Running From Volatile Memory" on page 2–45. However, in this configuration, the processor does not perform the initial boot loading operation. The EPCS flash memory stores the FPGA hardware image and the application image. During system power up, the FPGA configures itself from EPCS memory. Then the Nios II processor resets control to a small FPGA memory resource in the EPCS memory controller, and executes a small boot loader application that copies the application from EPCS memory to the application's run-time location.

To make this configuration work, you must instantiate the EPCS device controller core in your system hardware. Add the component using SOPC Builder.

Prepare for BSP configuration by performing the following steps:

1. **Nios II processor reset address—**Ensure that the Nios II processor's reset address is in the EPCS memory controller. Configure this option using SOPC Builder.
2. **Text section linker setting—**Ensure that the `.text` section maps to a volatile region of system memory.
3. **Other sections linker setting—**Ensure that all of the other sections, including the `.rodata` section, map to volatile memory.
4. **HAL C run-time configuration settings—**Use the following HAL C run-time configuration settings:
   - **hal.linker.allow_code_at_reset** `0`
   - **hal.linker.enable_alt_load** `0`
   - **hal.linker.enable_alt_load_copy_rwdata** `0`
   - **hal.linker.enable_alt_load_copy_exceptions** `0`
   - **hal.linker.enable_alt_load_copy_rodata** `0`

Booting and Running From FPGA Memory

In this configuration, the program is loaded in and runs from internal FPGA memory resources. The FPGA memory resources are automatically configured when the FPGA device is configured, so no additional boot loading operations are required.

Prepare for BSP configuration by performing the following steps:

1. **Nios II processor reset address—**Ensure that the Nios II processor's reset address is in the FPGA internal memory. Configure this option using SOPC Builder.
2. **Text section linker setting—**Ensure that the `.text` section maps to the internal FPGA memory.
3. **Other sections linker setting—**Ensure that all of the other sections map to the internal FPGA memory.

4. **HAL C run-time configuration settings**—Use the following HAL C run-time configuration settings:
   - **hal.linker.allow_code_at_reset** `1`
   - **hal.linker.enable_alt_load** `0`
   - **hal.linker.enable_alt_load_copy_rwdata** `0`
   - **hal.linker.enable_alt_load_copy_exceptions** `0`
   - **hal.linker.enable_alt_load_copy_rodata** `0`

This configuration requires that you generate FPGA memory HEX files for compilation to the FPGA image. This step is described in the following section.

Generating and Programming System Memory Images

After you configure your linker settings and boot loader configuration and build the application image **.elf** file, you must create a memory programming file. The flow for creating the memory programming file depends on your choice of FPGA, flash, or EPCS memory.

The easiest way to generate the memory files for your system is to use the application-generated makefile targets. The available **mem_init.mk** targets are listed in the "Creating Memory Initialization Files" section in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. You can also perform the same process manually, as shown in the following sections.

Generating memory programming files is not necessary if you want to download and run the application on the target system, for example, during the development and debug cycle.

Programming FPGA Memory

If your software application is designed to run from an internal FPGA memory resource, you must convert the application image **.elf** file to one or more HEX memory files. The Quartus II software compiles these HEX memory files to an FPGA image (**.sof**). When this image is loaded in the FPGA it initializes the internal memory blocks.

To create a HEX memory file from your **.elf** file, type the following command:

```
elf2mem --infile=<myapp>.elf --ptf=<system>.ptf ↵
```

This command creates one or more HEX memory files from application image <*myapp*>**.elf**, based on the SOPC Builder hardware description file <*system*>**.ptf**.

Compile the HEX memory files to an FPGA image using the Quartus II software. Initializing FPGA memory resources requires some knowledge of SOPC Builder and the Quartus II software.

Configuring and Programming Flash Memory

After you configure and build your BSP project and your application image **.elf** file, you must generate a flash programming file. The **nios2-flash-programmer** tool uses this file to configure the flash memory device through a programming cable, such as the USB-Blaster cable.

Creating a Flash Image File

If a boot loader application is required in your system, then you must first create a flash image file for your system. This section shows some standard commands to create a flash image file. The section does not address the case of programming and configuring the FPGA image from flash memory.

The following standard commands create a flash image file for your flash memory device:

- **Boot loader required and EPCS flash device used—**To create an EPCS flash device image, type the following command:

  ```
  elf2flash --epcs --after=<standard>.flash --input=<myapp>>.elf \
  --output=<myapp>.flash ↵
  ```

  This command converts the application image in the file <*myapp*>**.elf** to a flash record format, and creates the new file <*myapp*>**.flash** that contains the new flash record appended to the FPGA hardware image in <*standard*>**.flash**.

- **Boot loader required and CFI flash memory used—**To create a CFI flash memory image, type the following command:

  ```
  elf2flash --base=0x0 --reset=0x0 --end=0x1000000 \
            --boot=<boot_loader_cfi>.srec \
            --input=<myapp>.elf --output=<myapp>.flash ↵
  ```

  This command converts the application image in the file <*myapp*>**.elf** to a flash record format, and creates the new file <*myapp*>**.flash** that contains the new flash record appended to the CFI boot loader in <*boot_loader_cfi*>**.srec**. The flash record is to be downloaded to the reset address of the Nios II processor, `0x0`, and the base address of the flash device is `0x0`. If you use the Altera-supplied boot loader, your user-created program sections are also loaded from the flash memory to their run-time locations.

- **No boot loader required and CFI flash memory used—**To create a CFI flash memory image, if no boot loader is required, type the following command:

  ```
  elf2flash --base=0x0 --reset=0x0 --end=0x1000000 \
            --input=<myapp>.elf --output=<myapp>.flash ↵
  ```

  This command and its effect are almost identical to those of the command to create a CFI flash memory image if a boot loader is required. In this case, no boot loader is required, and therefore the `--boot` command-line option is not present.

The Nios II EDS includes two precompiled boot loaders for your use, one for CFI flash devices and another for EPCS flash devices. The source code for these boot loaders can be found in the **<***nios2eds dir***>/components/altera_nios2/boot_loader_sources/** directory.

Programming Flash Memory

The easiest way to program your system flash memory is to use the application-generated makefile target called **program-flash**. This target automatically downloads the flash image file to your development board through a JTAG download cable. You can also perform this process manually, using the **nios2-flash-programmer** utility. This utility takes a flash file and some command line arguments, and programs your system's flash memory. The following command-line examples illustrate use of the **nios2-flash-programmer** utility to program your system flash memory:

- **Programming CFI Flash Memory**—To program CFI flash memory with your flash image file, type the following command:

  ```
  nios2-flash-programmer --base=0x0 <myapp>.flash ↵
  ```

  This command programs a flash memory located at base address `0x0` with a flash image file called **<myapp>.flash**.

- **Programming EPCS Flash Memory**—To program EPCS flash memory with your flash image file, type the following command:

  ```
  nios2-flash-programmer --epcs --base=0x0 <myapp>.flash ↵
  ```

  This command programs an EPCS flash memory located at base address `0x0` with a flash image file called *<myapp>***.flash**.

The **nios2-flash-programmer** utility requires that your FPGA has already been configured with your system hardware image. You must download your **.sof** file with the **nios2-configure-sof** command before running the **nios2-flash-programmer** utility.

For more information about how to configure, program, and manage your flash memory devices, refer to the *Nios II Flash Programmer User Guide*.

Conclusion

Altera recommends that you use the Nios II software build tools flow for hardware designs that contain a Nios II processor. This chapter provides information about the Nios II software build tools flow that complements the *Nios II Software Developer's Handbook*. It discusses recommended design practices and implementation information, and provides pointers to related topics for more in-depth information.

Referenced Documents

This chapter references the following documents:

- *AN391: Profiling Nios II Systems*
- *AN458: Alternative Nios II Boot Methods*
- *AN459: Guidelines for Developing a Nios II HAL Device Driver*
- *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*
- *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*
- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *Ethernet and the TCP/IP Networking Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*
- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*

- *HAL API Reference* appendix of the *Nios II Software Developer's Handbook*
- *Hardware Acceleration and Coprocessing* chapter of the *Embedded Design Handbook*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Nios II C2H Compiler User Guide*
- *Nios II Flash Programmer User Guide*
- *Nios II Hardware Development Tutorial*
- *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Developer's Handbook*
- *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*
- *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*
- *Scatter-Gather DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *System ID Core* chapter in volume 5 of the *Quartus II Handbook*
- *Using Nios II Floating-Point Custom Instructions*
- *Using Nios II Tightly Coupled Memory Tutorial*
- *Using the Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 2–1 shows the revision history for this chapter.

Table 2–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release.	—

© June 2008 Altera Corporation

3. Debugging Nios II Designs

ED51003-1.1

This chapter 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 chapter includes the following sections that discuss debugging techniques and tools to address difficult embedded design problems:

- "Debuggers"
- "Run-Time Analysis Debug Techniques" on page 3–11

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:

- "Nios II Software Development Tools"
- "FS2 Console" on page 3–9
- "SignalTap II Embedded Logic Analyzer" on page 3–10
- "Lauterbach Trace32 Debugger and PowerTrace Hardware" on page 3–10
- "Insight and Data Display Debuggers" on page 3–11

Nios II Software Development Tools

The Nios II Integrated Development Environment (IDE) is a graphical user interface (GUI) that supports creating, modifying, building, running, and debugging Nios II programs. The Nios II software build tools are command-line utilities available from a Nios II command shell. Using the software build tools provides fine control over the build process and project settings, but also requires more expertise than does using the Nios II IDE.

SOPC Builder 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. SOPC Builder 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.

This section contains the following subsections, which describe the debugging tools and support features available in the Nios II software development tools:

- "Nios II System ID"
- "Project Templates" on page 3–3
- "Configuration Options" on page 3–3
- "Nios II GDB Console and GDB Commands" on page 3–5
- "Nios II Terminal Window and stdio Library Functions" on page 3–6

- "Importing Projects Created Using the Nios II Software Build Tools" on page 3–7
- "Selecting a Processor Instance in a Multiple Processor Design" on page 3–7
- "Debugging the Lauterbach PowerTrace to Nios II Processor Connection" on page 3–10
- "C Source Correlation" on page 3–11

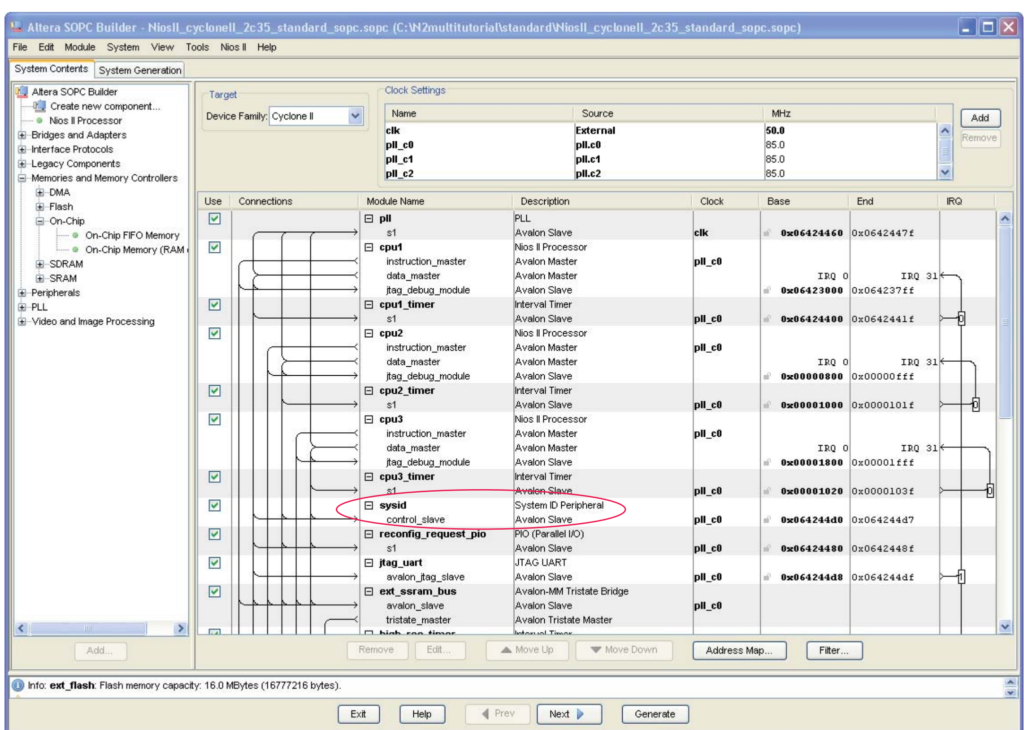
Nios II System ID

The system ID feature is available as a system component in SOPC Builder. The component allows the debugger to identify attempts to download software projects with system libraries that were generated for a different SOPC Builder system. This feature protects you from inadvertently using an executable and loadable 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, always select **Validate Nios II system ID before software download** on the **Main** tab of the Nios II IDE **Debug** dialog box, as shown in Figure 3–4 on page 3–8.

The system ID feature requires that the SOPC Builder design include a system ID component. Figure 3–1 shows an SOPC Builder system with a system ID component.

Figure 3–1. SOPC Builder System With System ID Component



For more information about the System ID component, refer to the *System ID Core* chapter in volume 5 of the *Quartus II Handbook*.

Project Templates

The Nios II IDE helps you to create a simple, small, and pretested software project to test a new board.

The Nios II IDE provides a mechanism to create new software projects using project templates. To create a new project for which you already have source code, perform the following steps:

1. In the Nios II C/C++ perspective, on the File menu, on the New submenu, click **Nios II C/C++ Application**.

   The New Project wizard for Nios II C/C++ application projects appears, pre-selecting the current SOPC Builder system **.ptf** file.

2. Click **Next**.
3. In the Select Project Template list, click **Blank Project**.
4. If your project contains multiple Nios II processors, in the CPU list, click the CPU you wish to run this application software.
5. Click **Finish**.
6. On the **Nios II IDE C/C++ Projects** page, copy your source code files to the new project by dragging them onto the newly created project label.

To create a simple test program to test a new board, perform these steps with the following exceptions:

- In step 3, click **Hello World Small**
- Do not perform step 6.

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.

Configuration Options

The following Nios II IDE configuration options increase the amount of debugging information available for your application image **.elf** file:

- Objdump File
- Show Make Commands
- Show Line Numbers

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. Example 3–1 shows part of the C and assembly code section of an **.objdump** file for the Nios II built-in Hello World Small project.

Example 3–1. Piece of Code in .objdump File From Hello World Small Project

```
06000170 <main>:

include "sys/alt_stdio.h"

int main()
{
6000170:deffff04 addisp,sp,-4
alt_putstr("Hello from Nios II!\n");
6000174:01018034 movhir4,1536
6000178:2102ba04 addir4,r4,2792
600017c:dfc00015 stwra,0(sp)
6000180:60001c00 call60001c0 <alt_putstr>
6000184:003fff06 br6000184 <main+0x14>

06000188 <alt_main>:
* the users application, i.e. main().
*/

void alt_main (void)
{
6000188:deffff04 addisp,sp,-4
600018c:dfc00015 stwra,0(sp)

static ALT_INLINE void ALT_ALWAYS_INLINE
alt_irq_init (const void* base)
{
NIOS2_WRITE_IENABLE (0);
6000190:000170fa wrctlienable,zero
NIOS2_WRITE_STATUS (NIOS2_STATUS_PIE_MSK);
6000194:00800044 movir2,1
6000198:1001703a wrctlstatus,r2
```

To enable this option in the Nios II IDE, perform the following steps:

1. On the Window menu, click **Preferences**.
2. On the list to the left, click **Nios II**.
3. On the **Nios II** page, turn on **Generate objdump file**.

   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 the Nios II IDE, perform the following steps:

1. On the Window menu, click **Preferences**.
2. On the list to the left, click **Nios II**.
3. On the **Nios II** page, turn on **Show command lines when running 'make' (i.e. Don't use '-s' flag on make)**.

Show Line Numbers

To enable display of C source-code line numbers in the Nios II IDE, 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**.

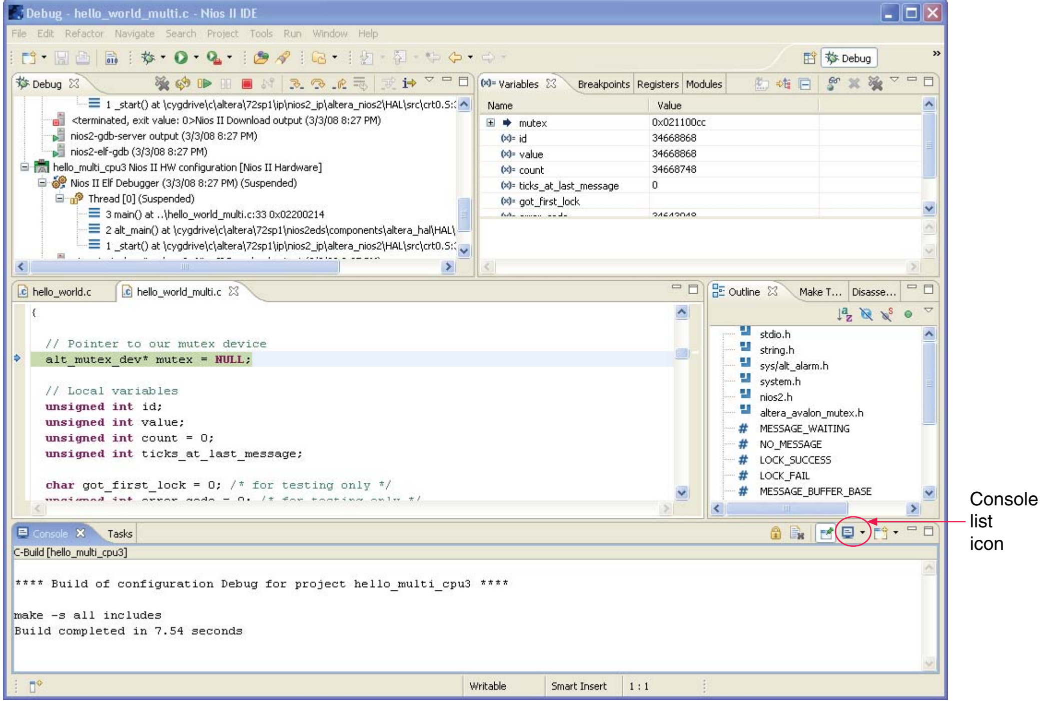
Nios II GDB Console and GDB Commands

The Nios II GDB console allows you to send GDB commands to the Nios II processor directly.

First, enable the GDB console on the **Debugger** tab of the **Debug** dialog box, by turning on **Verbose console mode**. This mode displays all of the GDB commands that are sent to and received by the Nios II processor on the GDB console.

To display this console, which allows you to view these commands and to enter your own GDB commands, click the blue monitor icon on the lower right corner of the Nios II Debug perspective. 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 Nios II GDB console. Figure 3–2 shows the console list icon—the blue monitor icon and black arrow—that allow you to display the GDB console.

Figure 3–2. Console List Icon



An example of a useful command you can enter in the Nios II 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 www.expertcomsoft.com.

Nios II Terminal Window and stdio Library Functions

If the Nios II processor outputs characters using the `stdio` library functions, but no terminal session exists to receive these characters, the Nios II software system deadlocks. If you use the `alt_log()` function, rather than the `printf()` function, to transmit characters to a nios2-terminal session or to the Nios II IDE terminal window, the system does not deadlock if no terminal session is available to receive the transmitted characters.

If neither of the consoles is connected, the output buffer fills. Then the system hangs on the next `stdio` library function write. If you select the **Reduced Device Drivers** option on the **System Properties** page in SOPC Builder, `stdout` uses the polling-mode device driver. This driver polls in a loop, waiting for the character output buffer to empty before the driver can transmit more characters. If no real-time operating system is running, and the E_WOULD_BLOCK `ioctl()` control code is not sent to the UART driver for the Nios II terminal, the Nios II software system hangs waiting to transmit characters as the result of a `printf()` statement in application code.

For more information about the `alt_log()` function, refer to *AN459: Guidelines for Developing a Nios II HAL Device Driver*.

Importing Projects Created Using the Nios II Software Build Tools

Whether a project is created and built using the Nios II software build tools or using the Nios II IDE, you can debug the resulting **.elf** image file in the Nios II IDE.

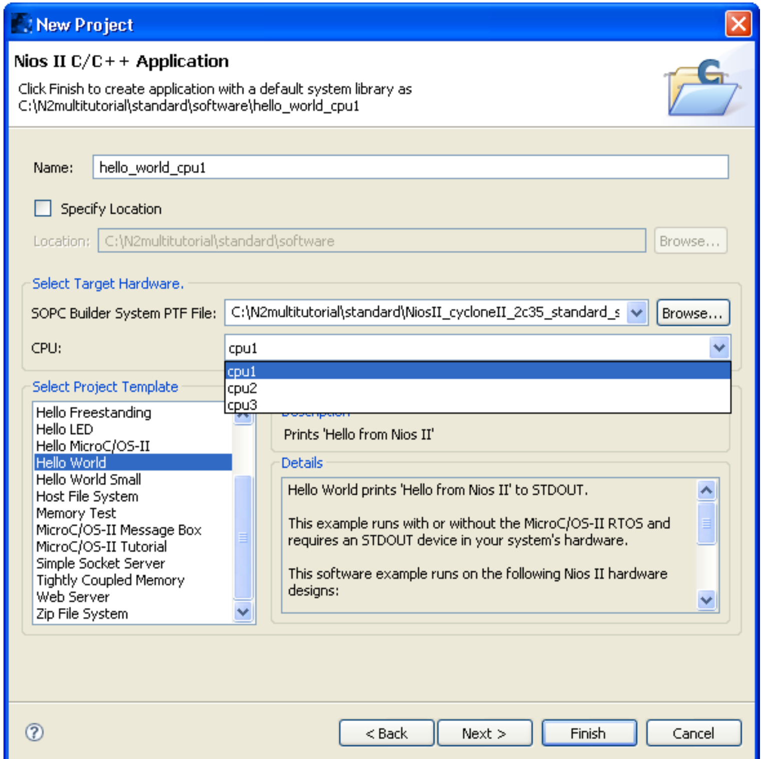
For information about how to import a project created with the Nios II software build tools to the Nios II IDE, refer to the "Getting Started" section in the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

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 the application project, the Nios II IDE generates a system library. For system library generation, you must specify the CPU to which the application project is targeted.

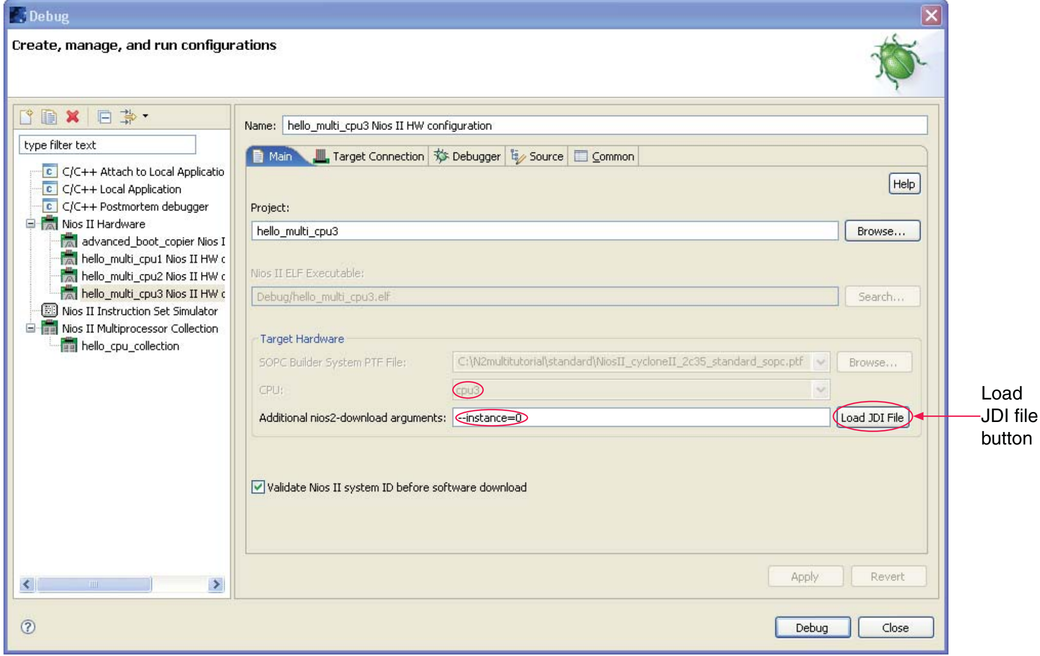
Figure 3–3 shows how you specify the CPU for the application in the Nios II IDE. The **Nios II C/C++ Application** page of the New Project wizard collects the information required for system library creation. This page derives the list of available CPU choices from the **.ptf** for the system.

Figure 3–3. Nios II IDE Nios II C/C++ Application Page — CPU Selection



In the **Main** tab of the **Debug** dialog box, shown in Figure 3–4, click the **Load JDI File** button to select the JTAG debug interface (**.jdi**) file for your SOPC Builder project. The **.jdi** file is typically located in the same directory as the SRAM object file (**.sof**) for the project. The **.jdi** file is parsed and its contents compared to the name of the CPU you select for the current project, to determine the correct instance ID number. The command-line option `--instance = `*<instance ID>* is appended to the implicit debug command that the Nios II IDE runs. The text for the command-line option appears in the **Additional nios2-download arguments** field next to the **Load JDI File** button. Clicking this button ensures that the proper instance ID is used for the selected CPU, whether or not the Quartus II software modified the instance IDs.

Figure 3–4. Nios II IDE Debug Configuration Page — Load JDI File Button



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. Example 3–2 shows the system response to a **jtagconfig -n** command.

Example 3–2. Two-FPGA System Response to jtagconfig Command

```
[SOPC Builder]$ jtagconfig -n
1) USB-Blaster [USB-0]
 120930DD EP2S60
 Node 11104600
 Node 0C006E00
2) USB-Blaster [USB-1]
 020B40DD EP2C35
 Node 11104601
 Node 11104602
 Node 11104600
 Node 0C006E00
```

The response in Example 3–2 lists two different FPGAs, connected to the running JTAG server through different USB-Blaster™ cables. The cable attached to the USB-0 port is connected to a JTAG node in an SOPC Builder subsystem with a single Nios II core. The cable attached to the USB-1 port is connected to a JTAG node in an SOPC Builder subsystem with three Nios II cores. The node numbers represent JTAG nodes inside the FPGA. The appearance of the node number `0x111046xx` 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 `111`. 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.

FS2 Console

On Windows platforms, you can use a Nios II-compatible version of the First Silicon Solutions, Inc. (FS2) console. The FS2 console is very helpful for low-level system debug, especially when bringing up a system or a new board. It provides a TCL-based scripting environment and many features for testing your system, from low-level register and memory access to debugging your software (trace, breakpoints, and single-stepping).

To run the FS2 console in the Nios II IDE, on the **Debugger** tab of the **Debug** dialog box, turn on **Use FS2 console window for trace and watchpoint support**. To run the FS2 console using the software build tools, use the **nios2-console** command.

For more details about the Nios II-compatible version of the FS2 console, refer to the FS2-provided documentation in your Nios II installation, at **$SOPC_KIT_NIOS2\bin\fs2\doc**.

In the FS2 console, the **sld info** command returns information about the JTAG nodes connected to the system-level debug (SLD) hubs—one SLD hub per FPGA—in your system. If you receive a failure response, refer to the FS2-provided documentation for more information.

Use the **sld info** command to verify your system configuration. After communication is established, you can perform simple memory reads and writes to verify basic system functionality. The FS2 console can write bytes or words, if Avalon® Memory-Mapped (Avalon-MM) interface `byteenable` signals are present. In contrast, the Nios II IDE memory window can perform only 32-bit reads and writes regardless of the 8- or 16-bit width settings for the values retrieved. If you encounter any issues, you can perform these reads and writes and capture SignalTap® II embedded logic analyzer traces of related hardware signals to diagnose a hardware level problem in the memory access paths.

SignalTap II Embedded Logic Analyzer

The SignalTap II embedded logic analyzer can help you to catch some software-related problems, such as an interrupt service routine that does not properly clear the interrupt signal.

For information about the SignalTap II embedded logic analyzer, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook* and *AN323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems*, and the *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook*.

The Nios II plug-in for the SignalTap II embedded logic analyzer enables you to capture a Nios II processor's program execution.

For more information about the Nios II plug-in for the SignalTap II embedded logic analyzer, refer to *AN446: Debugging Nios II Systems with the SignalTap II Logic Analyzer*.

Lauterbach Trace32 Debugger and PowerTrace Hardware

Lauterbach Datentechnik GmBH (Lauterbach) (www.lauterbach.com) provides the Trace32 ICD-Debugger for the Nios II processor. The product contains both hardware and software. In addition to a connection for the 10-pin JTAG connector that is used for the Altera USB-Blaster cable, the PowerTrace hardware has a 38-pin mictor connection option.

Lauterbach also provides a module for off-chip trace capture. For more information, refer to the downloadable *Nios II Debugger and Trace* document (file name **debugger_nios.pdf**) on the Lauterbach website (www.lauterbach.com). This document is also available in the latest distribution of the Lauterbach Trace32 software. If the document does not appear in your Lauterbach Trace32 installation directory, under **PDF**, Altera recommends that you download the latest distribution of the software from the Lauterbach website. Currently, this document is also available from the Lauterbach website Support section, under Update Online Manuals, as a separate PDF file for download.

Lauterbach also provides an instruction-set simulator for Nios II systems.

The *Nios II Debugger and Trace* document from Lauterbach contains important information about the order in which devices must be powered up. 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.

Debugging the Lauterbach PowerTrace to Nios II Processor Connection

A script is available for diagnosing difficulties with the Lauterbach PowerTrace connection and execution of the Trace32 **System.Up** command.

This script is available in the Altera online solutions database. Go to the support center at www.altera.com and click **Browse Support Solutions,** or in the Altera website Search field, type `rd03052008_529` and click **Search**.

C Source Correlation

A script is available for setting up software paths so that the Lauterbach Trace32 debugger can match source code locations to the loaded **.elf** file contents. This mapping enables the Trace32 software to display source code, and enables you to set breakpoints in the displayed C source code files.

This script is available in the Altera online solutions database. Go to the support center at www.altera.com and click **Browse Support Solutions**, or in the Altera website Search field, type `rd03052008_123` and click **Search**.

Registering Trace Signals

Trace signals must have uniform timing. Uniform timing can be achieved by ensuring uniform length traces on the board, or by registering the output signals.

A solution that includes descriptions for using a single PLL and for registering trace signals is available from the Lauterbach website. Refer to the downloadable online *Nios II Instantiating the Off-chip Trace Logic* document (file name **app_nios.pdf**) on the Lauterbach website (www.lauterbach.com). Currently, this document is available from the Lauterbach website Support section, under Update Online Manuals, as a separate PDF file for download.

Insight and Data Display Debuggers

The Tcl/Tk-based Insight GDB GUI installs with the Altera-specific GNU GDB distribution that is part of the Nios II Embedded Design Suite (EDS). To launch the Insight debugger from the Nios II command shell, type the following command:
`nios2-debug <file>.elf` ↵

Although the Insight debugger has fewer features than the Nios II IDE, this debugger supports faster communication between host and target, and therefore provides a more responsive debugging experience.

Another alternative debugger is the Data Display Debugger (DDD). This debugger is compatible with GDB commands—it is a user interface to the GDB debugger—and can therefore be used to debug Nios II software designs. The DDD can display data structures as graphs.

Run-Time Analysis Debug Techniques

This section discusses methods and tools available to analyze a running software system.

Software Profiling

Altera provides the following tools to profile the run-time behavior of your software system:

- **GNU profiler**—The Nios II EDS toolchain includes the **gprof** utility for profiling your application. This method of profiling reports how long various functions run in your application.

- **High resolution timer**—The SOPC Builder 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 SOPC Builder 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 SOPC Builder timer peripheral, refer to the *Timer Core* chapter in volume 5 of the *Quartus II Handbook*, and to the *Developing Nios II Software* chapter of the *Embedded Design Handbook*.

For additional information about the SOPC Builder performance counter peripheral, refer to the *Performance Counter Core* chapter in volume 5 of the *Quartus II Handbook*.

Watchpoints

Watchpoints provide a powerful method to capture all writes to a global variable that appears to be corrupted. The Nios II IDE supports watchpoints directly or through the FS2 console. Before you can set watchpoints in the Nios II IDE directly, you must make sure that, on the **Debugger** tab of the **Debug** dialog box, **Use FS2 console window for trace and watchpoint support** is turned off.

For more information about watchpoints, refer to the Nios II online Help. In the Nios II IDE, on the Help menu, click **Search**. In the search field, type `watchpoint`, and select the topic **Working with breakpoints and watchpoints**.

To enable watchpoints, you must configure the Nios II processor's debug level in SOPC Builder to debug level 2 or higher. To configure the Nios II processor's debug level in SOPC Builder to the appropriate level, perform the following steps:

1. On the SOPC Builder **System Contents** tab, 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 3–5 on page 3–14.
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 3–5 on page 3–14 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 in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

Stack Overflow

You can enable the Nios II IDE to check for stack overflow. On the **System Properties** configuration page of your system library project, turn on **Run time stack checking**. 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.

Hardware Abstraction Layer (HAL)

The Altera HAL provides the interfaces and resources required by the device drivers for most SOPC Builder system peripherals. You can customize and debug these drivers for your own SOPC Builder system. To learn more about debugging HAL device drivers and SOPC Builder peripherals, refer to *AN459: Guidelines for Developing a Nios II HAL Device Driver*.

Breakpoints

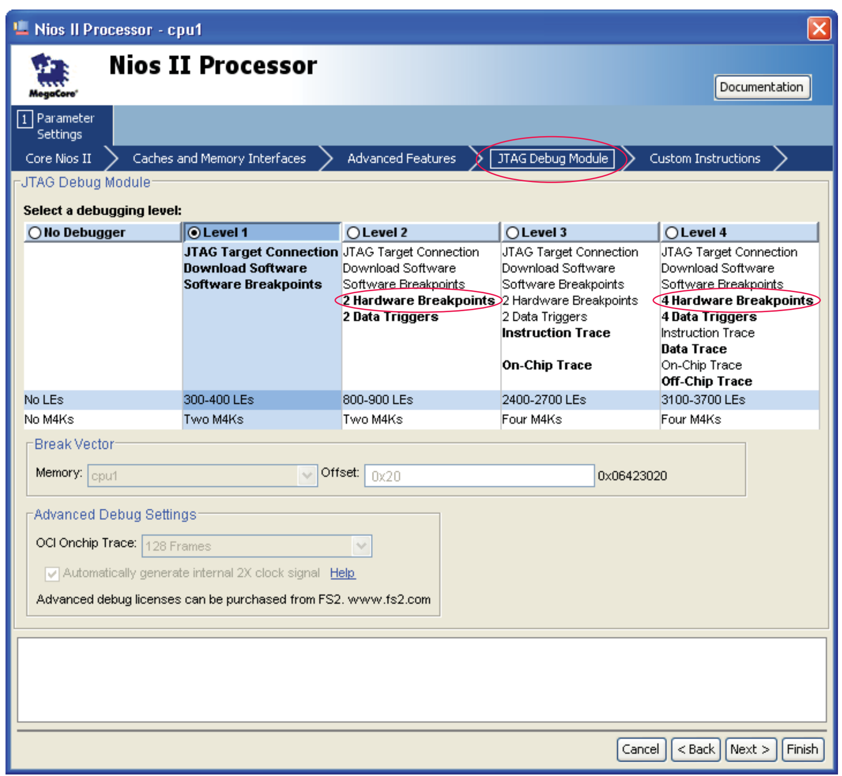
You can set hardware breakpoints on code located in read-only memory such as flash memory. If you set a breakpoint in a read-only area of memory, a hardware breakpoint, rather than a software breakpoint, is selected automatically.

To enable hardware breakpoints, you must configure the Nios II processor's debug level in SOPC Builder to debug level 2 or higher. To configure the Nios II processor's debug level in SOPC Builder to the appropriate level, perform the following steps:

1. On the SOPC Builder **System Contents** tab, 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 3–5.
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. Figure 3–5 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 3–5. Nios II Processor — JTAG Debug Module — SOPC Builder Configuration Page



For more information about the Nios II processor debug levels, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

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 IDE, perform the following steps:

1. In the Nios II C/C++ perspective, right-click your application project. A list of options appears.
2. On the list, click **Properties**.
3. In the left pane, click **C/C++ Build**.
4. Under Configuration Settings, click the **Tool Settings** tab.
5. On the list to the left, under **Nios II Compiler**, click **General**.
6. In the **Optimization Levels** list, click **None (-O0)**.

To set this switch in the Nios II software build tools flow, modify the application makefile to assign `APP_CFLAGS_OPTIMIZATION := -O0`.

Conclusion

Successful debugging of Nios II designs requires expertise in board layout, FPGA configuration, and Nios II software tools and application software. Altera and third-party tools are available to help you debug your Nios II application. This chapter describes debugging techniques and tools to address difficult embedded design problems.

Referenced Documents

This chapter references the following documents:

- *AN323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems*
- *AN391: Profiling Nios II Systems*
- *AN446: Debugging Nios II Systems with the SignalTap II Logic Analyzer*
- *AN459: Guidelines for Developing a Nios II HAL Device Driver*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Developing Nios II Software* chapter of the *Embedded Design Handbook*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Performance Counter Core* chapter in volume 5 of the *Quartus II Handbook*
- *System ID Core* chapter in volume 5 of the *Quartus II Handbook*
- *Timer Core* chapter in volume 5 of the *Quartus II Handbook*
- *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook*

Document Revision History

Table 3–1 shows the revision history for this chapter.

Table 3–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release.	—

ALTERA®

4. Nios II Command-Line Tools

ED51004-2.0

Introduction

This chapter describes the Nios® II command-line tools that are provided with the Nios II Embedded Development Suite (EDS). The chapter describes both the Altera® tools and the GNU tools. Most of the commands are located in the **$SOPC_KIT_NIOS2\bin** and **$SOPC_KIT_NIOS2\sdk2** subdirectories of your Nios II EDS installation.

The Altera command line tools are useful for a range of activities, from board and system-level debugging to programming an FPGA configuration file (**.sof**). For these tools, the examples expand on the brief descriptions of the Altera-provided command-line tools for developing Nios II programs in the *Altera-Provided Development Tools* chapter of the *Nios II Software Developer's Guide*. 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 chapter are available in the Nios II command shell. For most of the commands, you can obtain help in this shell by typing

```
<command name> --help ↵
```

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

```
$SOPC_KIT_NIOS2/sdk_shell ↵
```

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

Altera Command-Line Tools for Board Bringup and Diagnostics

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

jtagconfig

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

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

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

jtagconfig Usage Example

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

1. Open a Nios II command shell.
2. In the command shell, type the following command:
   `jtagconfig -n` ↵

Example 4–1 shows a typical system response to the `jtagconfig -n` command.

Example 4–1. jtagconfig Example Response

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

The information in the response varies, depending on the particular FPGA, its configuration, and the JTAG connection cable type. Table 4–1 describes the information that appears in the response in Example 4–1.

Table 4–1. Interpretation of jtagconfig Command Response

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

The device name is read from the text file **pgm_parts.txt** in your Quartus® II installation. In Example 4–1, the name is EP1S40/_HARDCOPY_FPGA_PROTOTYPE because the silicon identification number on the JTAG chain for the FPGA device is `020050DD`, which maps to the names EP1S40<device-specific name>, a couple of which end in the string _HARDCOPY_FPGA_PROTOTYPE. The internal nodes are nodes on the system-level debug (SLD) hub. All JTAG communication to an Altera FPGA passes through this hub, including advanced debugging capabilities such as the SignalTap® II embedded logic analyzer and the debugging capabilities in the Nios II Integrated Development Environment (IDE).

Example 4–1 illustrates a single cable connected to a single-device JTAG chain. However, your computer can have multiple JTAG cables, connected to different systems. Each of these systems can have multiple devices in its JTAG chain. Each device can have multiple JTAG debug modules, JTAG UART modules, and other kinds of JTAG nodes. Use the `jtagconfig -n` command to help you understand the devices with JTAG connections to your host PC and how you can access them.

nios2-configure-sof

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

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

nios2-configure-sof Usage Example

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

1. Open a Nios II command shell.
2. In the command shell, change to the directory in which your **.sof** is located. By default, the correct location is the top-level Quartus II project directory.
3. In the command shell, type the following command:
   `nios2-configure-sof` ↵

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

system-console

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

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

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

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

For System Console usage examples and a comprehensive list of system console commands, refer to the *System Console User Guide*. On-line training is available at http://www.altera.com/training.

Altera Command-Line Tools for Hardware Development

This section describes Altera command-line tools useful for hardware project development. They are useful for all projects created with SOPC Builder, whether or not the project includes a Nios II processor.

quartus_cmd and sopc_builder

These commands create scripts that automate generation of SOPC Builder systems and compilation of the corresponding Quartus II projects.

You can use these commands to create a flow that maintains only the minimum source files required to build your Quartus II project. If you copy an existing project to use as the basis for development of a new project, you should copy only this minimum set of source files. Similarly, when you check in files to your version control system, you want to check in only the minimum set required to reconstruct the project.

To reconstruct an SOPC Builder system, the following files are required:

- <project>**.qpf** (Quartus II project file)
- <project>**.qsf** (Quartus II settings file)
- <SOPC Builder system>**.sopc** (SOPC Builder system description)
- The additional HDL, BDF, or BSF files in your existing project

If you work with the hardware design examples that are provided with the Quartus II installation, Altera recommends that you copy each set of source files to a working directory to avoid modifying the original source files inadvertently. Run the script on the new working directory.

To create a flow that maintains only the minimum source files, perform the following steps:

1. Copy the required source files to a working directory, maintaining a correct copy of each source file elsewhere.
2. Change to this working directory.
3. To generate a **.sof** to configure your FPGA, type the following command sequence:

```
sopc_builder --no_splash -s --generate ↵
quartus_cmd <project>.qpf -c <project>.qsf ↵
```

The shell script in Example 4–2 illustrates these commands. This script automates the process of generating SOPC Builder systems and compiling Quartus II projects across any number of subdirectories. The script is an example only, and may require modification for your project. If you want to compile the Quartus II projects, set the `COMPILE_QUARTUS` variable in the script to `1`.

Example 4–2. Script to Generate SOPC Builder System and Compile Quartus II Projects (Part 1 of 2)

```
#!/bin/sh
COMPILE_QUARTUS=0
#
# Resolve TOP_LEVEL_DIR, default to PWD if no path provided.
#
if [ $# -eq 0 ]; then
   TOP_LEVEL_DIR=$PWD
else
   TOP_LEVEL_DIR=$1
fi
echo "TOP_LEVEL_DIR is $TOP_LEVEL_DIR"
echo
#
# Generate SOPC list...
#
SOPC_LIST=`find $TOP_LEVEL_DIR -name "*.sopc"`
#
# Generate Quartus II project list.
#
PROJ_LIST=`find $TOP_LEVEL_DIR -name "*.qpf" | sed s/\.qpf//g`
#
# Main body of the script.  First "generate" all of the SOPC Builder
# systems that are found, then compile the Quartus II projects.
#
#
# Run SOPC Builder to "generate" all of the systems that were found.
#
for SOPC_FN in $SOPC_LIST
do
   cd `dirname $SOPC_FN`
   if [ ! -e `basename $SOPC_FN .sopc`.vhd -a ! -e `basename $SOPC_FN .sopc`.v ]; then
      echo; echo
      echo "INFO:  Generating $SOPC_FN SOPC Builder system."
      sopc_builder -s --generate=1 --no_splash
      if [ $? -ne 4 ]; then
         echo; echo
         echo "ERROR:  SOPC Builder generation for $SOPC_FN has failed!!!"
         echo "ERROR:  Please check the SOPC file and data " \
                   "in the directory `dirname $SOPC_FN` for errors."
      fi
   else
      echo; echo
      echo "INFO:  HDL already exists for $SOPC_FN, skipping Generation!!!"
   fi
   cd $TOP_LEVEL_DIR
done
#
# Continued...
#
```

Example 4–2. Script to Generate SOPC Builder System and Compile Quartus II Projects (Part 2 of 2)

```
#
# Now, generate all of the Quartus II projects that were found.
#
if [ $COMPILE_QUARTUS ]; then
    for PROJ in $PROJ_LIST
    do
        cd `dirname $PROJ`
        if [ ! -e `basename $PROJ`.sof ]; then
            echo; echo
            echo "INFO:  Compiling $PROJ Quartus II Project."
            quartus_cmd `basename $PROJ`.qpf -c `basename $PROJ`.qsf
            if [ $? -ne 4]; then
                echo; echo
                echo "ERROR:  Quartus II compilation for $PROJ has failed!!!."
                echo "ERROR:  Please check the Quartus II project " \
                                "in `dirname $PROJ` for details."
            fi
        else
            echo; echo
            echo "INFO:  SOF already exists for $PROJ, skipping compilation."
        fi
        cd $TOP_LEVEL_DIR
    done
fi
```

The commands and script in Example 4–2 are provided for example purposes only. Altera does not guarantee the functionality for your particular use.

Altera Command-Line Tools for Flash Programming

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

When you use the Nios II IDE to program flash memory, the Nios II IDE 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 IDE and command-line usage of the Nios II Flash Programmer and related tools, refer to the *Nios II Flash Programmer User Guide*.

nios2-flash-programmer

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

nios2-flash-programmer Usage Example

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

1. Follow the steps in "nios2-download" on page 4–9, or use the Nios II IDE, 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 SOPC Builder. 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 the section "elf2flash, bin2flash, and sof2flash".

4. Type the following command to program the resulting **.flash** file in the CFI device:

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

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

   `nios2-download -g -r`↵

elf2flash, bin2flash, and sof2flash

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

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

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

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

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

Example 4–3 shows a sample auto-generated script.

Example 4–3. Sample Auto-Generated Script:

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

cd <full path to your project>/Debug

# Creating .flash file for the FPGA configuration
#"$SOPC_KIT_NIOS2/bin/sof2flash" --offset=0x400000 --input="full path to your SOF" \
                --output="<your design>.flash"

# Programming flash with the FPGA configuration
#"$SOPC_KIT_NIOS2/bin/nios2-flash-programmer" --base=0x00000000 --sidp=0x00810828 \
                --id=1436046714 --timestamp=1169569475 --instance=0 "<your design>.flash"
#
# Creating .flash file for the project
"$SOPC_KIT_NIOS2/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
"$SOPC_KIT_NIOS2/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
"$SOPC_KIT_NIOS2/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
"$SOPC_KIT_NIOS2/bin/nios2-flash-programmer" --base=0x00000000 --sidp=0x00810828 \
                --id=1436046714 --timestamp=1169569475 --instance=0 "<filename>.flash"
```

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

bin2flash Usage Example

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

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

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

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

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

Altera Command-Line Tools for Software Development and Debug

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

nios2-terminal

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

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

nios2-download

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

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

nios2-download Usage Example

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

1. Open a Nios II command shell.
2. Change to the directory in which your **.elf** file is located. If you use the Nios II IDE for development, the correct location is often the **Debug** or **Release** subdirectory of your top-level project.
3. In the command shell, type the following command to download and start your program:
   `nios2-download -g <`*project name*`>.elf` ↵
4. Optionally, use the `nios2-terminal` command to connect to view any output or provide any input to the running program.

nios2-stackreport

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

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

Example 4–4 illustrates the information this command provides.

Example 4–4. nios2-stackreport Command and Response

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

nios2-stackreport Usage Example

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

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

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

validate_zip

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

validate_zip Usage Example

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

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

   `validate_zip <file>.zip` ↵

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

nios2-ide

On Linux and Windows systems, you can type `nios2-ide` in a command shell to launch the Nios II IDE. On Windows systems, you can also use the Nios II IDE launch icon in SOPC Builder.

The `nios2-ide` command does not call the executable file directly. Instead, it runs a simple Bourne shell wrapper script, which calls the **nios2-ide** executable file. The Linux and Windows platform versions of the wrapper script follow.

Linux wrapper script

```
#!/bin/sh
# This is the linux-gtk version of the nios2-ide launcher script
# set the default workspace location for linux
WORKSPACE="$HOME/nios2-ide-workspace-7.2"
WORKSPACE_ARGS="-data $WORKSPACE"
# if -data is already passed in, we can't specify it
# again when calling nios2-ide
for i in $*
do
  if [ "x$i" = "x-data" ]; then
```

```
        WORKSPACE_ARGS=""
  fi
done
exec $SOPC_KIT_NIOS2/bin/eclipse/nios2-ide -configuration
$HOME/.nios2-ide-6.1 $WORKSPACE_ARGS "$@"
```

Windows wrapper script

```
#!/bin/sh
# This is the win32 version of the nios2-ide launcher script
# It simply invokes the binary with the same arguments as
# passed in.
# By doing this, the user will default to the same workspace as
# when launched using the Windows shortcut, as "persisted"
# in the configuration/.settings/org.eclipse.ui.ide.prefs file.
cd "$SOPC_KIT_NIOS2/bin/eclipse"
exec ./nios2-ide-console "$@"
```

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 IDE debugger session, you should stop the **nios2-gdb-server.exe** process on Windows platforms, or type the following command on Linux platforms:
`pkill -9 -f nios2-gdb-server` ↵

nios2-gdb-server Usage Example

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

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

1. Open a Nios II command shell.
2. In the command shell, type the following command to start the GDB server on the machine that is connected through a JTAG interface to the Nios II system you wish to debug:
   `nios2-gdb-server --tcpport 2342 --tcppersist` ↵

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

   Following is the system response:

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

   Now you can connect to your server (locally or remotely) and start debugging.
3. Type the following command to start a GDB client that targets your **.elf** file:
   `nios2-elf-gdb` <*file*>`.elf` ↵

Example 4–5 shows a sample session.

Example 4–5. 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.

nios2-debug

This command is a wrapper around the Tcl/Tk-based Insight GDB GUI, which installs with the Altera-specific GNU GDB distribution that is part of the Nios II EDS.

The command-line option `-save-gdb-script` saves the session script, and the option `-command=`<*GDB script name*> restores a previous GDB session by executing its previously saved GDB script. Use this option to restore break and watch points.

For more information about the Insight GDB GUI, refer to the Insight documentation available at sources.redhat.com.

nios2-debug Usage Example

After you generate the **.elf** file manually or using the Nios II IDE, perform the following steps to open an Insight debugger session:

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 IDE for development, the correct location is often the **Debug** or **Release** subdirectory of your top-level project.

3. In the command shell, type the following command:

   `nios2-debug <file>.elf ↵`

   Your **.elf** file is parsed and downloaded to memory in your Nios II subsystem, and the main debugger window opens, with the first executable line in the `main()` function highlighted. This debugger window displays your Insight debugging session. Simply click on the **Continue** menu item to run your code, or set some breakpoints to experiment.

Altera Command-Line Nios II Software Build Tools

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

Unlike the Nios II IDE-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 *Introduction to the Nios II Software Build Tools* and the *Using the Nios II Software Build Tools* chapters of the *Nios II Software Developer's Handbook*. The **software_examples** directory for each current Nios II development board contains examples that use the GNU make-based software build flow. The examples for your development board are located in the following location:

`$SOPC_KIT_NIOS2/examples/[verilog|vhdl]/<dev_board>/`
`<design>/software_examples`

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 *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook.*

For information on the many additional options available to you in the Nios II software build tools, refer to the *Introduction to the Nios II Software Build Tools*, *Using the Nios II Software Build Tools*, and *Nios II Software Build Tools Reference* chapters of the *Nios II Software Developer's Handbook*, and the *Developing Nios II Software* chapter of the *Embedded Design Handbook*.

BSP Related Tools

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

- `nios2-bsp-create-settings` creates a BSP settings file.
- `nios2-bsp-update-settings` updates a BSP settings file.
- `nios2-bsp-query-settings` queries an existing BSP settings file.
- `nios2-bsp-generate-files` generates all the files related to a given BSP settings file.

- `nios2-bsp` is a script that includes most of the functionality of the preceding commands.
- `create-this-bsp` is a high-level script that creates a BSP for a specific hardware design example.

Application Related Tools

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

- `nios2-app-generate-makefile` creates a makefile for your application.
- `nios2-lib-generate-makefile` creates a makefile for your application library.
- `nios2-c2h-generate-makefile` creates a makefile fragment for the C2H compiler.
- `create-this-app` is a high-level script that creates an application for a specific hardware design example.

GNU Command-Line Tools

The Nios II GCC toolchain contains the GNU Compiler Collection, the GNU binutils, and the newlib C library. You can follow links to detailed documentation from the Nios II EDS documentation launchpad found in your Nios II EDS distribution. To start the launchpad on Windows platforms, on the Start menu, click **All Programs**. On the All Programs menu, on the Altera submenu, on the Nios II EDS *<version>* submenu, click **Literature**. On Linux platforms, run the program in the file **$SOPC_KIT_NIOS2/documents/index.htm**. In addition, more information about the GNU GCC toolchain is available on the World Wide Web.

nios2-elf-addr2line

This command returns a source code line number for a specific memory address. The command is similar to but more specific than the `nios2-elf-objdump` command described in “nios2-elf-objdump” on page 4–21 and the `nios2-elf-nm` command described in “nios2-elf-nm” on page 4–20.

Use the `nios2-elf-addr2line` command to help validate code that should be stored at specific memory addresses. Example 4–6 illustrates its usage and results:

Example 4–6. nios2-elf-addr2line Utility Usage Example

```
[SOPC Builder]$ nios2-elf-addr2line --exe=<your project>.elf 0x1000020
${SOPC_KIT_NIOS2}/components/altera_nios2/HAL/src/alt_exception_entry.S:99
```

 © November 2008 Altera Corporation

nios2-elf-addr2line Usage Example

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

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

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

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

nios2-elf-gdb

This command is a GDB client that provides a simple shell interface, with built-in commands and scripting capability. A typical use of this command is illustrated in the section “nios2-gdb-server” on page 4–11.

nios2-elf-readelf

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

nios2-elf-readelf Usage Example

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

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

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

Example 4–7 shows a search for the `http_read_line()` function in a **.elf** file.

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

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

Table 4–2 lists the meanings of the individual columns in Example 4–7.

Table 4–2. Interpretation of nios2-elf-readelf Command Response

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

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

nios2-elf-ar

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

nios2-elf-ar Usage Example

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

1. Open a Nios II command shell.
2. Change to the directory in which your object files are located.
3. In the command shell, type the following command:
```
nios2-elf-ar q <archive_name>.a <object files>
```

Example 4–8 shows how to create an archive of all of the object files in your current directory. In Example 4–8, the `q` option directs the command to append each object file it finds to the end of the archive. After the archive file is created, it can be distributed for others to use, and included as an argument in linker commands, in place of a long object file list.

Example 4–8. nios2-elf-ar Command Response

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

Linker

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

Linker Usage Example

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

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

The exact linker command line to link your executable may differ. When you build a project in the Nios II IDE, you can see the command line used to link your application. To turn on this option in the Nios II IDE, 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.

 © November 2008 Altera Corporation

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

nios2-elf-size

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

nios2-elf-size Usage Example

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

1. Open a Nios II command shell.
2. Change to the directory in which your **.elf** file is located.
3. In the command shell, type the following command:
   `nios2-elf-size` <*project*>`.elf`

Example 4–9 shows the size information this command provides.

Example 4–9. nios2-elf-size Command Usage

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

nios2-elf-strings

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

nios2-elf-strings Usage Example

The command has a single required argument:

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

nios2-elf-strip

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

nios2-elf-strip Usage Example

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

nios2-elf-strip Usage Notes

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

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

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

nios2-elf-gdbtui

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

The syntax for the `nios2-elf-gdbtui` command is identical to that for the `nios2-elf-gdb` command described in "nios2-elf-gdb" on page 4–15.

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

nios2-elf-gprof

This command allows you to profile your Nios II system.

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

nios2-elf-insight

The `nios2-debug` command described in "nios2-debug" on page 4–12 uses this command to start an Insight debugger session on the supplied **.elf** file.

nios2-elf-gcc and g++

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

Compilation Command Usage Example

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

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

More Complex Compilation Example

Example 4–10 is a Nios II IDE-generated command line that compiles C code in multiple files in many directories.

Example 4–10. Example nios2-elf-gcc Command Line

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

nios2-elf-c++filt

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

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

nios2-elf-c++filt Usage Example

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

nios2-elf-c++filt -n <*symbol name*> ↵

For example,

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

More Complex nios2-elf-c++filt Example

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

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

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

nios2-elf-nm

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

nios2-elf-nm Usage Example

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

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

More Complex nios2-elf-nm Example

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

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

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

nios2-elf-objcopy

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

Though typical usage converts from or to ELF files, the `objcopy` command is not restricted to conversions from or to ELF files. You can use this command to convert from, and to, any of the formats listed in Table 4–3.

Table 4–3. -objcopy Binary Formats

Command (...-objcopy)	Comments
elf32-littlenios2, elf32-little	Header little endian, data little endian, the default and most commonly used format
elf32-bignios2, elf32-big	Header big endian, data big endian
srec	S-Record (SREC) output format
symbolsrec	SREC format with all symbols listed in the file header, preceding the SREC data
tekhex	Tektronix hexadecimal (TekHex) format
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 on the World Wide Web. As of the initial publication of this handbook, you can refer to the www.sbprojects.com knowledge-base entry on file formats.

nios2-elf-objcopy Usage Example

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

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

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

```
nios2-elf-objcopy --help ↵
```

nios2-elf-objdump

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

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

nios2-elf-objdump Usage Description

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

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

nios2-elf-ranlib

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

For further information about this command, refer to "nios2-elf-ar" on page 4–16 or type `nios2-elf-ar --help` in a Nios II command shell.

Referenced Documents

This chapter references the following documents:

- *Altera-Provided Development Tools* chapter of the *Nios II Software Developer's Guide*
- *AN 391: Profiling Nios II Systems*
- *Developing Nios II Software* chapter of the *Embedded Design Handbook*
- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- Nios II Flash Programmer User Guide
- Nios II Software Build Tools Reference chapter of the *Nios II Software Developer's Handbook*
- Nios II Software Developer's Handbook
- *System Console User Guide*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook*

Document Revision History

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

Table 4–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2008 v2.0	Add System Console.	Add System Console.
March 2008 v1.0	Initial release.	—

© November 2008 Altera Corporation

5. Optimizing Nios II C2H Compiler Results

ED51005-1.1

Introduction

The Nios® II C2H Compiler is a powerful tool that generates hardware accelerators for software functions. The C2H Compiler enhances design productivity by allowing you to use a compiler to accelerate software algorithms in hardware. You can quickly prototype hardware functional changes in C, and explore hardware-software design tradeoffs in an efficient, iterative process. The C2H Compiler is well suited to improving computational bandwidth, as well as memory throughput. It is possible to achieve substantial performance gains with minimal engineering effort.

The structure of your C code affects the results you get from the C2H Compiler. Although the C2H Compiler can accelerate most ANSI C code, you might need to modify your C code to meet resource usage and performance requirements. This document describes how to improve the performance of hardware accelerators, by refactoring your C code with C2H-specific optimizations.

Prerequisites

To make effective use of this chapter, you should be familiar with the following topics:

- ANSI C syntax and usage
- Defining and generating Nios II hardware systems with SOPC Builder
- Compiling Nios II hardware systems with the Altera® Quartus® II development software
- Creating, compiling, and running Nios II software projects
- Nios II C2H Compiler theory of operation
- Data caching

To familiarize yourself with the basics of the C2H Compiler, refer to the *Nios II C2H Compiler User Guide*, especially the *Introduction to the C2H Compiler* and *Getting Started Tutorial* chapters. To learn about defining, generating, and compiling Nios II systems, refer to the *Nios II Hardware Development Tutorial.* To learn about Nios II software projects, refer to the *Nios II Software Development Tutorial*, available in the Nios II IDE help system. To learn about data caching, refer to the *Cache and Tightly-Coupled Memory* in the *Nios II Processor Reference Handbook*.

Cost and Performance

When writing C code for the C2H Compiler, you can optimize it relative to several optimization criteria. Often you must make tradeoffs between these criteria, which are listed below:

- Hardware cost—C2H accelerators consume hardware resources such as LEs, multipliers, and on-chip memory. This document uses the following terms to describe the hardware cost of C language constructs:
  - Free—the construct consumes no hardware resources.
  - Cheap—the construct consumes few hardware resources. The acceleration obtained is almost always worth the cost.
  - Moderate—the construct consumes some hardware resources. The acceleration obtained is usually worth the cost.
  - Expensive—the construct consumes substantial hardware resources. The acceleration obtained is sometimes worth the cost, depending on the nature of the application.
- Algorithm performance—A C2H accelerator performs the same algorithm as the original C software executed by a Nios II processor. Typically the accelerator uses many fewer clock cycles than the software implementation. This document describes the algorithm performance of C constructs as fast or slow. The concept of algorithm performance includes the concepts of latency and throughput. These concepts are defined under "Cycles Per Loop Iteration (CPLI)" on page 5–12.
- Hardware performance impact—Certain C language constructs, when converted to logic by the C2H Compiler, can result in long timing paths that can degrade f_{MAX} for the entire system or for the clock domain containing the C2H accelerator. This document clearly notes such situations and offers strategies for avoiding them.

Overview of the C2H Optimization Process

It is unlikely that you can meet all of your optimization goals in one iteration. Instead, plan on making the one or two optimizations that appear most relevant to your cost and performance issues. When you profile your system with the optimized accelerator, you can determine whether further optimizations are needed, and then you can identify the next most important optimization issue to address. By optimizing your accelerator one step at a time, you apply only the optimizations needed to achieve your goals.

Getting Started

The most important first step is to decide on a clear performance goal. Depending on your application, you may require a specific performance level from your algorithm. If you have already selected a target device, and if other hardware in the system is well defined, you might have specific hardware cost limitations. Alternatively, if you are in early phases of development, you might only have some general guidelines for conserving hardware resources. Finally, depending on your design needs and the f_{MAX} of your existing design, you might be concerned with possible f_{MAX} degradation. Refer to "Meeting Your Cost and Performance Goals" on page 5–3 for more information about cost and performance criteria.

The next step is to develop your algorithm in C, and, if possible, test it conventionally on the Nios II processor. This step is very helpful in establishing and maintaining correct functionality. If the Nios II processor is not fast enough for in-circuit testing of your unaccelerated algorithm, consider simulation options for testing.

When you are confident of your algorithm's correctness, you are ready to accelerate it. This first attempt provides a set of baseline acceleration metrics. These metrics help you assess the overall success of the optimization process.

Altera recommends that you maintain two copies of your algorithm in parallel: one accelerated and the other unaccelerated. By comparing the results of the accelerated and unaccelerated algorithms, you immediately discover any errors which you might inadvertently introduce while optimizing the code.

Iterative Optimization

The iteration phase of C2H Compiler optimization consists of these steps:

1. Profile your accelerated system.
2. Identify the most serious performance bottleneck.
3. Identify an appropriate optimization from the "Optimization Techniques" on page 5–14 section.
4. Apply the optimization and rebuild the accelerated system.

For instructions on profiling Nios II systems, refer to *AN391: Profiling Nios II Systems*.

Meeting Your Cost and Performance Goals

Having a clear set of optimization goals helps you determine when to stop optimization. Each time you profile your accelerated system, compare the results with your goals. You might find that you have reached your cost and performance goals even if you have not yet applied all relevant optimizations.

If your optimization goals are flexible, consider keeping track of your baseline acceleration metrics, and the acceleration metrics achieved at each optimization step. You might wish to stop if you reach a point of diminishing returns.

Factors Affecting C2H Results

This section describes key differences in the mapping of C constructs by a C compiler and the Nios II C2H Compiler. You must understand these differences to create efficient hardware.

C code originally written to run on a processor does not necessarily produce efficient hardware. A C compiler and the Nios II C2H Compiler both use hardware resources such as adders, multipliers, registers, and memories to execute the C code. However, while a C compiler assumes a sequential model of computing, the C2H Compiler assumes a concurrent model of computing. A C compiler maps C code to instructions which access shared hardware resources. The C2H Compiler maps C code to one or more state machines which access unique hardware resources. The C2H Compiler pipelines the computation as much as possible to increase data throughput. Example 5–1 illustrates this point.

Example 5–1. Pipelined Computation

```
int sumfunc(int a, int b, int c, int d)
{
int sum1 = a + b;
int sum2 = c + d;
int result = sum1 + sum2;
return result;
}
```

The `sumfunc()` function takes four integer arguments and returns their sum. A C compiler maps the function to three add instructions sharing one adder. The processor executes the three additions sequentially. The C2H Compiler maps the function to one state machine and three adders. The accelerator executes the additions for `sum1` and `sum2` concurrently, followed by the addition for `result`. The addition for `result` cannot execute concurrently with the `sum1` and `sum2` additions because of the data dependency on the `sum1` and `sum2` variables.

Different algorithms require different C structures for optimal hardware transformation. This chapter lists possible optimizations to identify in C code. Each C scenario describes the best methods to refactor the C code. The "Optimization Techniques"section discusses how to address the following potential problem areas:

- Memory Accesses and Variables
- Arithmetic and Logical Operations
- Statements
- Control Flow
- Subfunction Calls
- Resource Sharing
- Data Dependencies
- Memory Architecture

Memory Accesses and Variables

Memory accesses can occur when your C code reads or writes the value of a variable. Table 5–1 provides a summary of the key differences in the mapping of memory accesses between a C compiler and the C2H Compiler.

A C compiler generally allocates many types of variables in your data memory. These include scalars, arrays, and structures that are local, static, or global. When allocated in memory, variables are relatively cheap due to the low cost per bit of memory (especially external memory) and relatively slow due to the overhead of load or store instructions used to access them. In some situations, a C compiler is able to use processor registers for local variables. When allocated in processor registers, these variables are relatively fast and expensive.

The C2H Compiler allocates local scalar variables in registers implemented with logic elements (LEs), which have a moderate cost and are fast.

A C compiler maps pointer dereferences and array accesses to a small number of instructions to perform the address calculation and access to your data memory. Pointer dereferences and array accesses are relatively cheap and slow.

The C2H Compiler maps pointer dereferences and array accesses to a small amount of logic to perform the address calculation and creates a unique Avalon® Memory-Mapped (Avalon-MM) master port to access the addressed memory. This mapping is expensive due to the logic required to create an Avalon-MM master port. It is slow or fast depending on the type of memory connected to the port. Local arrays are fast because the C2H Compiler implements them as on-chip memories.

Table 5–1. Memory Accesses

C Construct	C Compiler Implementation	C2H Implementation
Local scalar variables	Allocated in memory (cheap, slow) or allocated in processor registers (expensive, fast)	Allocated in registers based on logic elements (LEs) (moderate cost, fast)
Uninitialized local array variables	Allocated in memory (cheap, slow)	Allocated in on-chip memory. (expensive, fast)
Initialized local array variables	Allocated in memory (cheap, slow)	Allocated in memory (cheap, slow)
All other types of variables	Allocated in memory (cheap, slow)	Allocated in memory (cheap, slow)
Pointer dereferences and nonlocal array accesses	Access normal data memory (cheap, slow)	Avalon-MM master port (expensive, slow or fast)

Arithmetic and Logical Operations

Table 5–2 provides a summary of the key differences in the mapping of arithmetic and logical operations between a C compiler and the C2H Compiler.

A C compiler maps arithmetic and logical operations into one or more instructions. In many cases, it can map them to one instruction. In other cases, it might need to call a function to implement the operation. An example of the latter occurs when a Nios II processor that does not have a hardware multiplier or divider performs a multiply operation.

The C2H Compiler implements the following logical operations simply as wires without consuming any logic at all.

- Shifts by a constant
- Multiplies and divides by a power of two constant
- Bitwise `AND`s and `OR`s by a constant

As a result, these operations are fast and free. The following is an example of one of these operations:

```
int result = some_int >> 2;
```

A C compiler maps this statement to a right shift instruction. The C2H Compiler maps the statement to wires that perform the shift.

Table 5–2. Arithmetic and Logical Operations

C Construct	C Compiler Implementation	C2H Implementation
Shift by constant or multiply or divide by power of 2 constant. *(1)* Example: `y = x/2;`	Shift instruction (cheap, fast)	Wires (free, fast)
Shift by variable Example: `y = x >> z;`	Shift instruction (cheap, fast)	Barrel shifter (expensive, fast)
Multiply by a value that is not a power of 2 (constant or variable Example: `y = x × z;`	Multiply operation (cheap, slow)	If the Quartus II software can produce an optimized multiply circuit (cheap, fast); otherwise a multiply circuit (expensive, fast)
Divide by a value that is not a power of 2 (constant or variable) Example: `y = x/z;`	Divide operation (cheap, slow)	Divider circuit (expensive, slow)
Bitwise AND or bitwise OR with constant Example: `y = x | 0xFFFF;`	AND or OR instruction (cheap, fast)	Wires (free, fast)
Bitwise AND or bitwise OR with variable Example: `y = x & z;`	AND or OR instruction (cheap, fast)	Logic (cheap, fast)

Notes to Table 5–2:

(1) Dividing by a *negative* power of 2 is expensive.

Statements

A C compiler maps long expressions (those with many operators) to instructions. The C2H Compiler maps long expressions to logic which could create a long timing path. The following is an example of a long expression:

```
int sum = a + b + c + d + e + f + g + h;
```

A C compiler creates a series of add instructions to compute the result. The C2H Compiler creates several adders chained together. The resulting computation has a throughput of one data transfer per clock cycle and a latency of one cycle.

A C compiler maps a large function to a large number of instructions. The C2H Compiler maps a large function to a large amount of logic which is expensive and potentially degrades f_{MAX}. If possible, remove from the function any C code that does not have to be accelerated.

A C compiler maps mutually exclusive, multiple assignments to a local variable as store instructions or processor register writes, which are both relatively cheap and fast. However, the C2H Compiler creates logic to multiplex between the possible assignments to the selected variable. Example 5–2 illustrates such a case.

Example 5–2. Multiple Assignments to a Single Variable

```
int result;
if (a > 100)
{
result = b;
}
else if (a > 10)
{
result = c;
}
else if (a > 1)
{
result = d;
}
```

A C compiler maps this C code to a series of conditional branch instructions and associated expression evaluation instructions. The C2H Compiler maps this C code to logic to evaluate the conditions and a three-input multiplexer to assign the correct value to `result`. Each assignment to `result` adds another input to the multiplexer. The assignments increase the amount of the logic, and might create a long timing path. Table 5–3 summarizes the key differences between the C compiler and C2H Compiler in handing C constructs.

Table 5–3. Statements

C Construct	C Compiler Implementation	C2H Implementation
Long expressions	Several instructions (cheap, slow)	Logic (cheap, degrades f_{MAX})
Large functions	Many instructions (cheap, slow)	Logic (expensive, degrades f_{MAX})
Multiple assignments to a local variable	Store instructions or processor register writes (cheap, fast)	Logic (cheap, degrades f_{MAX})

Control Flow

Table 5–4 provides a summary of the differences in the mapping of control flow between a C compiler and the C2H Compiler.

If Statements

The C2H compiler maps the expression of the `if` statement to control logic. The statement is controlled by the expression portion of the `if` statement.

Loops

Loops include `for` loops, `do` loops, and `while` loops. A C compiler and the C2H Compiler both treat the expression evaluation part of a loop just like the expression evaluation in an `if` statement. However, the C2H Compiler attempts to pipeline each loop iteration to achieve a throughput of one iteration per cycle. Often there is no overhead for each loop iteration in the C2H accelerator, because it executes the loop control concurrently with the body of the loop. The data and control paths pipelining allows the control path to control the data path. If the control path (loop expression) is dependent on a variable calculated within the loop, the throughput decreases because the data path must complete before control path can allow another loop iteration.

The expression, `while (++a < 10) { b++ };`runs every cycle because there is no data dependency. On the other hand, `while (a < 10) { a++ };` takes 2 cycles to run because the value of `<a>` is calculated in the loop.

A C compiler maps `switch` statements to the equivalent `if` statements or possibly to a jump table. The C2H Compiler maps `switch` statements to the equivalent `if` statements.

Table 5–4. Control Flow

C Construct	C Compiler Implementation	C2H Implementation
`if` statements	A few instructions (cheap, slow)	Logic (cheap, fast)
Loops	A few instructions of overhead per loop iteration (cheap, slow)	Logic (moderate, fast)
Switch statements	A few instructions (cheap, slow)	Logic (moderate, fast)
Ternary operation	A few instructions (cheap, slow)	Logic (cheap, fast)

Subfunction Calls

A C compiler maps subfunction calls to a few instructions to pass arguments to or from the subfunction and a few instructions to call the subfunction. A C compiler might also convert the subfunction into inline code. The C2H Compiler maps a subfunction call made in your top-level accelerated function into a new accelerator. This technique is expensive, and stalls the pipeline in the top-level accelerated function. It might result in a severe performance degradation.

However, if the subfunction has a fixed, deterministic execution time, the outer function attempts to pipeline the subfunction call, avoiding the performance degradation. In Example 5–3, the subfunction call is pipelined.

Example 5–3. Pipeline Stall

```
int abs(int a)
{
return (a < 0) ? -a : a;
}
int abs_sum(int* arr, int num_elements)
{
int i;
int result = 0;
for (i = 0; i < num_elements; i++)
{
result += abs(*arr++);
}
return result;
}
```

Resource Sharing

By default, the C2H Compiler creates unique instances of hardware resources for each operation encountered in your C code. If this translation consumes too many resources, you can change your C code to share resources. One mechanism to share resources is to use shared subfunctions in your C code. Simply place the code to be shared in a subfunction and call it from your main accelerated function. The C2H Compiler creates only one instance of the hardware in the function, shared by all function callers.

Example 5–4 uses a subfunction to share one multiplier between two multiplication operations.

Example 5–4. Shared Multiplier

```
int mul2(int x, int y)
{
  return x * y;
}
int muladd(int a, int b, int c, int d)
{
  int prod1 = mul2(a, b);
  int prod2 = mul2(c, d);
  int result = prod1 + prod2;
  return result;
}
```

Data Dependencies

A data dependency occurs when your C code has variables whose values are dependent on the values of other variables. Data dependency prevents a C compiler from performing some optimizations which typically result in minor performance degradation. When the C2H Compiler maps code to hardware, a data dependency causes it to schedule operations sequentially instead of concurrently, which can cause a dramatic performance degradation.

The algorithm in Example 5–5 shows data dependency.

Example 5–5. Data Dependency

```
int sum3(int a, int b, int c)
{
int sum1 = a + b;
int result = sum1 + c;
return result;
}
```

The C2H Compiler schedules the additions for `sum1` and `result` sequentially due to the dependency on `sum1`.

Memory Architecture

The types of memory and how they are connected to your system, including the C2H accelerator, define the memory system architecture. For many algorithms, appropriate memory architecture is critical to achieving high performance with the C2H Compiler. With an inappropriate memory architecture, an accelerated algorithm can perform more poorly than the same algorithm running on a processor.

Due to the concurrency possible in a C2H accelerator, compute-limited algorithms might become data-limited algorithms. To achieve the highest levels of performance, carefully consider the best memory architecture for your algorithm and modify your C code accordingly to increase memory bandwidth.

For the following discussion, assume that the initial memory architecture is a processor with a data cache connected to an off-chip memory such as DDR SDRAM.

The C code in Example 5–6 is data-limited when accelerated by the C2H Compiler because the `src` and `dst` dereferences both create Avalon-MM master ports that access the same Avalon-MM slave port. An Avalon-MM slave port can only handle one read or write operation at any given time; consequently, the accesses are interleaved, limiting the throughput to the memory bandwidth.

Example 5–6. Memory Bandwidth Limitation

```
void memcpy(char* dst, char* src, int num_bytes)
{
while (num_bytes-- > 0)
{
*dst++ = *src++;
}
}
```

The C2H Compiler is able to achieve a throughput of one data transfer per clock cycle if the code is modified and the appropriate memory architecture is available. The changes required to achieve this goal are covered in "Efficiency Metrics" on page 5–11.

Data Cache Coherency

When a C2H accelerator accesses memory, it uses its own Avalon-MM master port, which bypasses the Nios II data cache. Before invoking the accelerator, if the data is potentially stored in cache, the Nios II processor must write it to memory, thus avoiding the typical cache coherency problem. This cache coherency issue is found in any multimaster system that lacks support for hardware cache coherency protocols.

When you configure the C2H accelerator, you choose whether or not the Nios II processor flushes the data cache whenever it calls the accelerated function. If you enable this option, it adds to the overhead of calling the accelerator and causes the rest of the C code on the processor to temporarily run more slowly because the data cache must be reloaded.

You can avoid flushing the entire data cache. If the processor never shares data accessed by the accelerator, it does not need to flush the data cache. However, if you use memory to pass data between the processor and the accelerator, as is often the case, it might be possible to change the C code running on the processor to use uncacheable accesses to the shared data. In this case, the processor does not need to flush the data cache, but it has slower access to the shared data. Alternatively, if the size of the shared data is substantially smaller than the size of the data cache, the processor only needs to flush the shared data before calling the accelerator.

Another option is to use a processor without a data cache. Running without a cache slows down all processor accesses to memory but the acceleration provided by the C2H accelerator might be substantial enough to result in the overall fastest solution.

DRAM Architecture

Memory architectures consisting of a single DRAM typically require modification to maximize C2H accelerator performance. One problem with the DRAM architecture is that memory performance degrades if accesses to it are nonsequential. Because the DRAM has only one port, multiple Avalon-MM master ports accessing it concurrently prevent sequential accesses by one Avalon-MM master from occurring.

The default behavior of the arbiter in an SOPC Builder system is round-robin. If the DRAM controller (such as the Altera Avalon SDRAM controller) can only keep one memory bank open at a time, the master ports experience long stalls and do not achieve high throughput. Stalls can cause the performance of any algorithm accelerated using the C2H Compiler to degrade if it accesses memory nonsequentially due to multiple master accesses or nonsequential addressing.

For additional information about optimizing memory architectures in a Nios II system, refer to the *Cache and Tightly-Coupled Memory* in the *Nios II Software Developer's Handbook*.

Efficiency Metrics

There are several ways to measure the efficiency of a C2H accelerator. The relative importance of these metrics depends on the nature of your application. This section explains each efficiency metric in detail.

Cycles Per Loop Iteration (CPLI)

The C2H report section contains a CPLI value for each loop in an accelerated function. The CPLI value represents the number of clock cycles each iteration of the loop takes to complete once the initial latency is overcome. The goal is to minimize the CPLI value for each loop to increase the data throughput. It is especially important that the innermost loop of the function have the lowest possible CPLI because it executes the most often.

The CPLI value does not take into account any hardware stalls that might occur. A shared resource such as memory stalls the loop if it is not available. If you nest looping structures the outer loops stall and, as a result, reduce the throughput of the outer loops even if their CPLI equals one. The "Optimization Techniques" on page 5–14 section offers methods for maximizing the throughput of loops accelerated with the C2H Compiler.

Optimizations that can help CPLI are as follows:

- Reducing data dependencies
- Reducing the system interconnect fabric by using the `connect_variable` pragma

f_{MAX} is the maximum frequency at which a hardware design can run. The longest register-to-register delay or critical path determines f_{MAX}. The Quartus II software reports the f_{MAX} of a design after each compilation.

Adding accelerated functions to your design can potentially affect f_{MAX} in two ways: by adding a new critical path, or by adding enough logic to the design that the Quartus II fitter fails to fit the elements of the critical path close enough to each other to maintain the path's previous delay. The optimizations that can help with f_{MAX} are as follows:

- Pipelined calculations
- Avoiding division
- Reducing system interconnect fabric by using the `connect_variable` pragma
- Reducing unnecessary memory connections to the Nios II processor

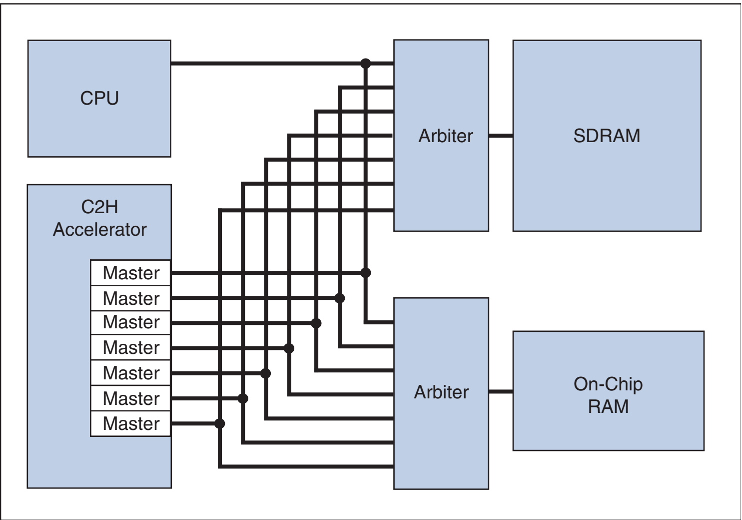
FPGA Resource Usage

Because an accelerated function is implemented in FPGA hardware, it consumes FPGA resources such as logic elements and memory. Sometimes, an accelerator consumes more FPGA resources than is desired or expected. Unanticipated resource usage has the disadvantage of consuming resources that are needed for other logic and can also degrade system f_{MAX}.

Avalon-MM Master Ports

The number of Avalon-MM master ports on the accelerator can heavily influence logic utilization. The C2H report, which the Nios II IDE displays after accelerating the function, reports how many Avalon-MM ports are generated. Multiple master ports can help increase the parallelization of logic when attached to separate memories. However, they have a cost in logic, and can also promote the creation of excessive arbitration logic when connected to the same memory port, as shown in Figure 5–1.

Figure 5–1. Too Many Master Ports



Embedded Multipliers

Multiplication logic is often available on the FPGA as dedicated hardware or created using logic elements. When you use dedicated hardware, be aware that having a large amount of multiplication logic can degrade the routing of your design because the fitter cannot place the multiplier columns to achieve a better fit. When creating multiplication logic from logic elements, be aware that this is expensive in resource usage, and can degrade f_{MAX}.

If one of the operands in a multiplication is a constant, the Quartus II software determines the most efficient implementation. Example 5–7 shows a optimization the Quartus II software might make:

Example 5–7. Quartus II Software Optimization for Multiplication by a Constant

```
/* C code mulitplication by a constant */
c = 7 × a;

/* Quartus II software optimization */
c = (4 × a) + (2 × a) + a;
```

Because the optimized equation includes multiplications by a constant factor of 2, the Quartus II software turns them into 2 shifts plus a add.

Embedded Memory

Embedded memory is a valuable resource for many hardware accelerators due to its high speed and fixed latency. Another benefit of embedded memory is that it can be configured with dual ports. Dual-ported memory allows two concurrent accesses to occur, potentially doubling the memory bandwidth. Whenever your code declares an uninitialized local array in an accelerated function, the C2H Compiler instantiates embedded memory to hold its contents. Use embedded memory only when it is appropriate; do not waste it on operations that do not benefit from its high performance.

Optimization tips that can help reduce FPGA resource use are:

- Using wide memory accesses
- Keeping loops rolled up
- Using narrow local variables

Data Throughput

Data throughput for accelerated functions is difficult to quantify. The Altera development tools do not report any value that directly corresponds to data throughput. The only true data throughput metrics reported are the number of clock cycles and the average number of clock cycles it takes for the accelerated function to complete. One method of measuring the data throughput is to use the amount of data processed and divide by the amount of time required to do so. You can use the Nios II processor to measure the amount of time the accelerator spends processing data to create an accurate measurement of the accelerator throughput.

Before accelerating a function, profile the source code to locate the sections of your algorithm that are the most time-consuming. If possible, leave the profiling features in place while you are accelerating the code, so you can easily judge the benefits of using the accelerator. The following general optimizations can maximize the throughput of an accelerated function:

- Using wide memory accesses
- Using localized data

For more information about profiling Nios II systems, refer to *Application Note 391: Profiling Nios II Systems*.

Optimization Techniques

Pipelining Calculations

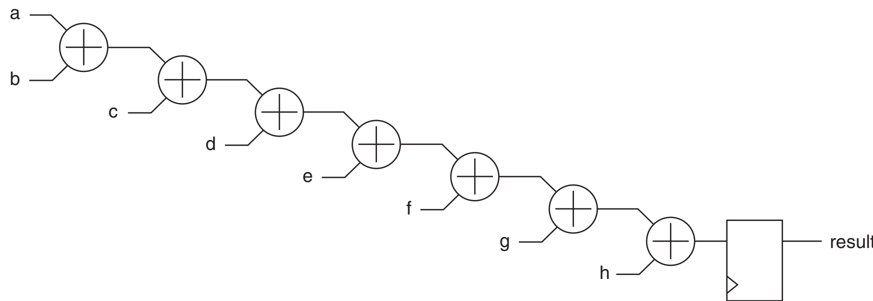
Although condensing multiple mathematical operations to a single line of C code, as in Example 5–8, can reduce the latency of an assignment, it can also reduce the clock speed of the entire design.

Example 5–8. Non-Pipelined Calculation (Lower Latency, Degraded f_{MAX})

```
int result = a + b + c + d + e + f + g + h;
```

Figure 5–2 shows the hardware generated for Example.

Figure 5–2. Non-Pipelined Calculations



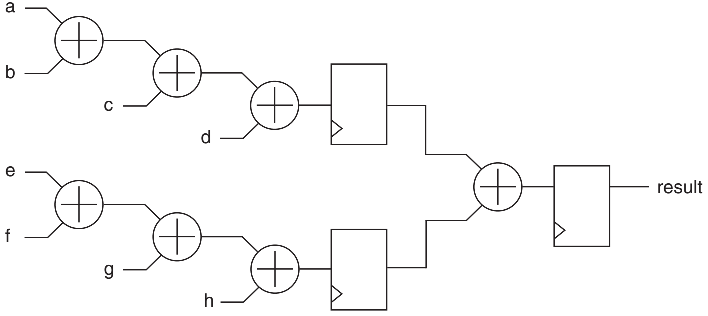
Often, you can break the assignment into smaller steps, as shown in Example 5–9. The smaller steps increase the loop latency, avoiding f_{MAX} degradation.

Example 5–9. Pipelined Calculation (Higher Latency, No f_{MAX} Degradation)

```
int result_abcd = a + b + c + d;
int result_efgh = e + f + g + h;
int result = result_abcd + result_efgh;
```

Figure 5–3 shows the hardware generated for Example 5–9.

Figure 5–3. Pipelined Calculations



Increasing Memory Efficiency

The following sections discuss coding practices that improve C2H performance.

Use Wide Memory Accesses

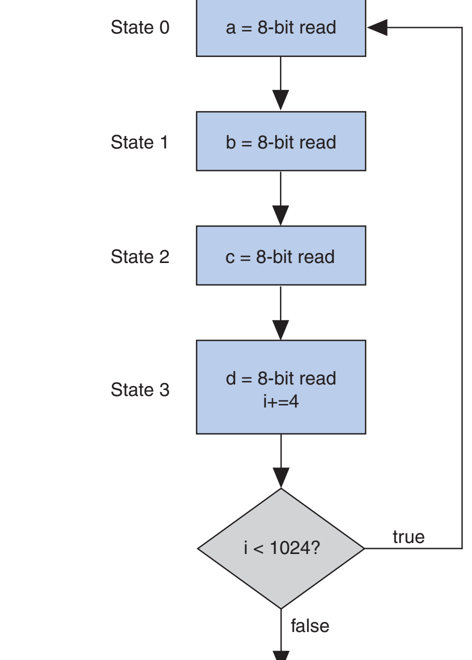
When software runs on a processor with a data cache, byte and halfword accesses to DRAM become full word transfers to and from the cache to guarantee efficient use of memory bandwidth. By contrast, when you make byte and halfword DRAM accesses in a C2H accelerator, as shown in Example 5–10, the Avalon-MM master port connected to the DRAM uses narrow accesses and fails to take advantage of the full data width of the memory.

Example 5–10. Narrow Memory Access (Slower Memory Access)

```
unsigned char narrow_array[1024];
char a, b, c, d;
for(i = 0; i < 1024; i+=4)
{
a = narrow_array[i];
b = narrow_array[i+1];
c = narrow_array[i+2];
d = narrow_array[i+3];
}
```

Figure 5–4 shows the hardware generated for Example 5–10.

Figure 5–4. Narrow Memory Access



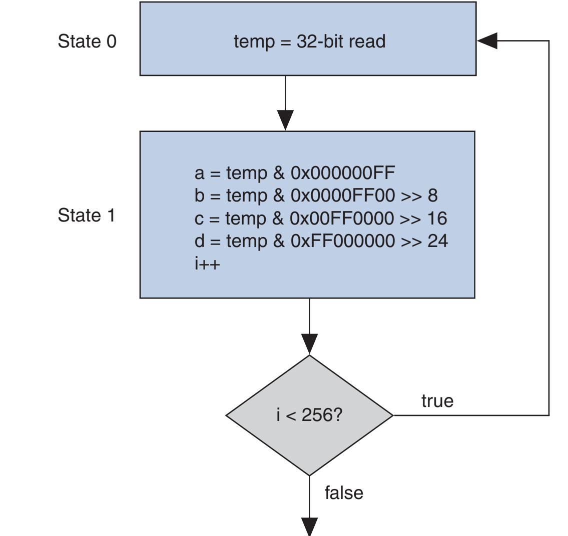
In a situation where multiple narrow memory accesses are needed, it might be possible to combine those multiple narrow accesses into a single wider access, as shown in Example 5–11. Combining accesses results in the use of fewer memory clock cycles to access the same amount of data. Consolidating four consecutive 8-bit accesses into one 32-bit access effectively increases the performance of those accesses by a factor of four.

Example 5–11. Wide Memory Access (Faster Memory Access)

```
unsigned int *wide_array = (unsigned int *) narrow_array;
unsigned int temp;
for(i = 0; i < 256; i++)
{
temp = wide_array[i];
  a = (char)( temp and 0x000000FF);
b = (char)( (temp and 0x0000FF00) >> 8);
c = (char)( (temp and 0x00FF0000) >> 16);
d = (char)( (temp and 0xFF000000) >> 24);
}
```

Figure 5–5 shows the hardware generated for Example 5–11.

Figure 5–5. Wide Memory Access



Segment the Memory Architecture

Memory segmentation is an important strategy to increase the throughput of the accelerator. Memory segmentation leads to concurrent memory access, increasing the memory throughput. There are multiple ways to segment your memory and the method used is typically application specific. Refer to Example 5–12 for the following discussions of memory segmentation optimizations.

Example 5–12. Memory Copy

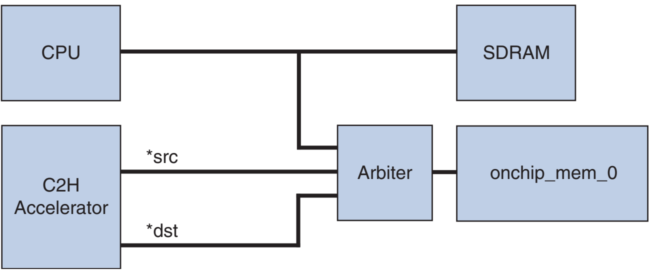
```
void memcpy(char* dst, char* src, int num_bytes)
{
while (num_bytes-- > 0)
{
*dst++ = *src++;
}
}
```

If the `src` and `dst` memory regions can be moved from the DRAM to an on-chip or off-chip SRAM, better performance is possible. To add on-chip memories, use SOPC Builder to instantiate an on-chip memory component (called **onchip_mem_0** in this example) with a 32-bit wide Avalon-MM slave port. Add the following pragmas to your C code before `memcpy`:

```
#pragma altera_accelerate connect_variable memcpy/dst to onchip_mem_0
#pragma altera_accelerate connect_variable memcpy/src to onchip_mem_0
```

The pragmas state that `dst` and `src` only connect to the **onchip_mem_0** component. This memory architecture offers better performance because SRAMs do not require large bursts like DRAMs to operate efficiently and on-chip memories operate at very low latencies. Figure Figure 5–6 shows the hardware generated for Example 5–12 with the data residing in on-chip RAM.

Figure 5–6. Use On-Chip Memory - Partition Memory for Better Bandwidth

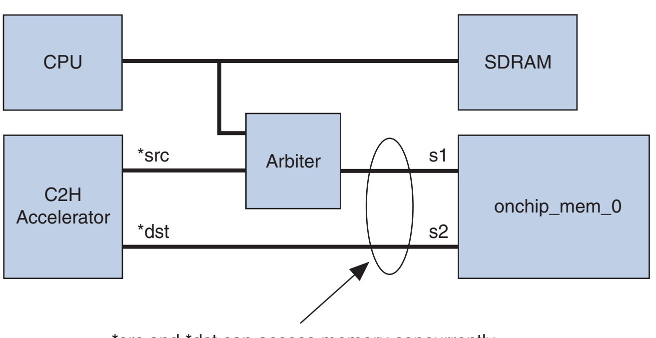


However, both master ports still share the single-port SRAM, **onchip_mem_0**, which can lead to a maximum throughput of one loop iteration every two clock cycles (a CPLI of 2). There are two solutions to this problem: Either create another SRAM so that each Avalon-MM master port has a dedicated memory, or configure the memories with dual-ports. For the latter solution, open your system in SOPC Builder and change the **onchip_mem_0** component to have two ports. This change creates slave ports called `s1` and `s2` allowing the connection pragmas to use each memory port as follows:

```
#pragma altera_accelerate connect_variable memcpy/dst to onchip_mem_0/s1
#pragma altera_accelerate connect_variable memcpy/src to onchip_mem_0/s2
```

These pragmas state that `dst` only accesses slave port `s1` of the **onchip_mem_0** component and that `src` only accesses slave port `s2` of the **onchip_mem_0** component. This new version of `memcpy` along with the improved memory architecture achieves a maximum throughput of one loop iteration per cycle (a CPLI of 1). Figure 5–7 shows the hardware generated for Example 5–12 with the data stored in dual-port on-chip RAM.

Figure 5–7. Use Dual-Port On-Chip Memory



Use Localized Data

Pointer dereferences and array accesses in accelerated functions always result in memory transactions though an Avalon-MM master port. Therefore, if you use a pointer to store temporary data inside an algorithm, as in Example 5–13, there is a memory access every time that temporary data is needed, which might stall the pipeline.

Example 5–13. Temporary Data in Memory (Slower)

```
for(i = 0; i < 1024; i++)
{
for(j = 0; j < 10; j++)
{/* read and write to the same location */
AnArray[i] += AnArray[i] * 3;
}
}
```

Often, storing that temporary data in a local variable, as in Example 5–14, increases the performance of the accelerator by reducing the number of times the accelerator must make an Avalon-MM access to memory. Local variables are the fastest type of storage in an accelerated function and are very effective for storing temporary data.

Although local variables can help performance, too many local variables can lead to excessive resource usage. This is a tradeoff you can experiment with when accelerating a function with the C2H Compiler.

Example 5–14. Temporary Data in Registers (Faster)

```
int temporary;
for(i = 0; i < 1024; i++)
{
temporary = AnArray[i]; /* read from location i */
for(j = 0; j < 10; j++)
{
    /* read and write to a registered value */
temporary += temporary * 3;
}
AnArray[i] = temporary; /* write to location i */
}
```

Reducing Data Dependencies

The following sections provide information on reducing data dependencies.

Use __restrict__

By default, the C2H Compiler cannot pipeline read and write pointer accesses because read and write operations may occur at the same memory location. If you know that the `src` and `dst` memory regions do not overlap, add the `__restrict__` keyword to the pointer declarations, as shown in Example 5–15.

Example 5–15. ***__restrict__ Usage***

```
void memcpy(char* __restrict__ dst, char* __restrict__ src, int
num_bytes)
{
while (num_bytes-- > 0)
{
*dst++ = *src++;
}
}
```

The `__restrict__` declaration on a pointer specifies that accesses via that pointer do not alias any memory addresses accessed by other pointers. Without `__restrict__`, the C2H Compiler must schedule accesses to pointers strictly as written which can severely reduce performance.

It is very important that you verify that your algorithm operates correctly when using `__restrict__` because this option can cause sequential code to fail when accelerated. The most common error is caused by a read and write pointer causing overlapping accesses to a dual port memory. You might not detect this situation when the function executes in software, because a processor can only perform one access at a time, however by using `__restrict__` you are allowing the C2H Compiler to potentially schedule the read and write accesses of two pointers to occur concurrently.

The most common type of data dependency is between scalar data variables. A scalar data dependency occurs when an assignment relies on the result of one or more other assignments. The C code in Example 5–16 shows a data dependency between `sum1` and `result`:

Example 5–16. Scalar Data Dependency

```
int sum3(int a, int b, int c)
{
int sum1 = a + b;
int result = sum1 + c;
return result;
}
```

A C compiler attempts to schedule the instructions to prevent the processor pipeline from stalling. There is no limit to the number of concurrent operations which you can exploit with the C2H Compiler. Adding all three integers in one assignment removes the data dependency, as shown in Example 5–17.

Example 5–17. Scalar Data Dependency Removed

```
int sum3(int a, int b, int c)
{
int result = a + b + c;
return result;
}
```

The other common type of data dependency is between elements of an array of data. The C2H Compiler treats the array as a single piece of data, assuming that all accesses to the array overlap. For example, the `swap01` function in Example 5–18 swaps the values at index 0 and index 1 in the array pointed to by *<p>*.

Example 5–18. Array Data Dependency

```
void swap01(int* p)
{
int tmp = p[0];
p[0] = p[1];
p[1] = tmp;
}
```

The C2H Compiler is unable to detect that the `p[0]` and `p[1]` accesses are to different locations so it schedules the assignments to `p[0]` and `p[1]` sequentially. To force the C2H Compiler to schedule these assignments concurrently, add `__restrict__` to the pointer declaration, as shown in Example 5–19.

Example 5–19. Array Data Dependency Removed

```
void swap01(int* p)
{
int* __restrict__ p0 = andp[0];
int* __restrict__ p1 = andp[1];
int tmp0 = *p0;
int tmp1 = *p1;
*p0 = tmp1;
*p1 = tmp0;
}
```

Now, the C2H Compiler attempts to perform the assignments to `p[0]` and `p[1]` concurrently. In this example, there is only a significant performance increase if the memory containing `p[0]` and `p[1]` has two or more write ports. If the memory is single-ported then one access stalls the other and little or no performance is gained.

A form of scalar or array data dependency is the in-scope data dependency. Example 5–20 exhibits an in-scope dependency, because it takes pointers to two arrays and their sizes and returns the sum of the contents of both arrays.

Example 5–20. In-Scope Data Dependency

```
int sum_arrays(int* arr_a, int* arr_b,
int size_a, int size_b)
{
int i;
int sum = 0;
for (i = 0; i < size_a; i++)
{
sum += arr_a[i];
}
for (i = 0; i < size_b; i++)
{
sum += arr_b[i];
}
return sum;
}
```

There is a dependency on the sum variable which causes C2H accelerator to execute the two loops sequentially. There is no dependency on the loop index variable <*i*> between the two loops because the algorithm reassigns <*i*> to 0 in the beginning of the second loop.

Example 5–21 shows a new version of `sum_arrays` that removes the in-scope dependency:

Example 5–21. In-Scope Data Dependency Removed

```
int sum_arrays(int* arr_a, int* arr_b,
int size_a, int size_b)
{
int i;
int sum_a = 0;
int sum_b = 0;
for (i = 0; i < size_a; i++)
{
sum_a += arr_a[i];
}
for (i = 0; i < size_b; i++)
{
sum_b += arr_b[i];
}
return sum_a + sum_b;
}
```

Using separate sum variables in each loop removes the in-scope dependency. The accelerator adds the two independent sums together at the end of the function to produce the final sum. Each loop runs concurrently although the longest loop determines the execution time. For best performance, connect `arr_a` and `arr_b` to a memory with two read ports or two separate memories.

Sometimes it is not possible to remove data dependencies by simple changes to the C code. Instead, you might need to use a different algorithm to implement the same functionality. In Example 5–22, the code searches for a value in a linked list and returns 1 if the value found and 0 if it is not found. Arguments to the search function are the pointer to the head of the linked list and the value to match against.

Example 5–22. Pointer-Based Data Dependency

```
struct item
{
int value;
struct item* next;
};
int search(struct item* head, int match_value)
{
struct item* p;
for (p=head; p != NULL; p=p->next)
{
if (p->value == match_value)
{
return 1; // Found a match
}
}
return 0;  // No match found
}
```

The C2H Compiler is not able to achieve a throughput of one comparison per cycle due to the `p=p->next <p>` does not occur until the next pointer location has been read from memory, causing a latency penalty to occur each time the loop state machine reaches this line of C code. To achieve better performance, use a different algorithm that supports more parallelism. For example, assume that the values are stored in an array instead of a linked list, as in Example 5–23. Arguments to the new search function are the pointer to the array, the size of the array, and the value to match against.

Example 5–23. Pointer-Based Data Dependency Removed

```
int search(int* arr, int num_elements, int match_value)
{
for (i = 0; i < num_elements; i++)
{
if (arr[i] == match_value)
{
return 1;  // Found a match
}
}
return 0;  // No match found
}
```

This new search function achieves a throughput of one comparison per cycle assuming there is no contention for memory containing the `arr` array. Prototype such a change in software before accelerating the code, because the change affects the functionality of the algorithm.

Reducing Logic Utilization

The following sections discuss coding practices you can adopt to reduce logic utilization.

Use "do-while" rather than "while"

The overhead of `do` loops is lower than those of the equivalent `while` and `for` loops because the accelerator checks the loop condition after one iteration of the loop has executed. The C2H Compiler treats a `while` loop like a `do` loop nested in an `if` statement. Example 5–24 illustrates code that the C2H Compiler transforms into a `do` loop and nested `if` statement.

Example 5–24. while Loop

```
int sum(int* arr, int num_elements)
{
int result = 0;
while (num_elements-- > 0)
{
result += *arr++;
}
return result;
}
```

Example 5–25 is the same function rewritten to show how the C2H Compiler converts a `while` loop to an `if` statement and a `do` loop.

Example 5–25. Converted while Loop

```
int sum(int* arr, int num_elements)
{
int result = 0;
if (num_elements > 0)
{
do
{
result += *arr++;
} while (--num_elements > 0);
}
return result;
}
```

Notice that an extra `if` statement outside the `do` loop is required to convert the `while` loop to a `do` loop. If you know that the `sum` function is never called with an empty array, that is, the initial value of `num_elements` is always greater than zero, the most efficient C2H code uses a `do` loop instead of the original `while` loop. Example 5–26 illustrates this optimization.

Example 5–26. do Loop

```
int sum(int* arr, int num_elements)
{
int result = 0;
do
{
result += *arr++;
} while (--num_elements > 0);
return result;
}
```

Use Constants

Constants provide a minor performance advantage in C code compiled for a processor. However, they can provide substantial performance improvements in a C2H accelerator.

Example 5–27 demonstrates a typical add and round function.

Example 5–27. Add and Round with Variable Shift Value

```
int add_round(int a, int b, int sft_amount)
{
int sum = a + b;
return sum >> sft_amount;
}
```

As written above, the C2H Compiler creates a barrel shifter for the right shift operation. If `add_round` is always called with the same value for `sft_amount`, you can improve the accelerated function's efficiency by changing the `sft_amount` function parameter to a `#define` value and changing all your calls to the function. Example 5–28 is an example of such an optimization.

Example 5–28. Add and Round with Constant Shift Value

```
#define SFT_AMOUNT 1
int add_round(int a, int b)
{
int sum = a + b;
return sum >> SFT_AMOUNT;
}
```

Alternatively, if `add_round` is called with a few possible values for `sft_amount`, you can still avoid the barrel shifter by using a `switch` statement which just creates a multiplexer and a small amount of control logic. Example 5–29 is an example of such an optimization.

Example 5–29. Add and Round with a Finite Number of Shift Values

```
int add_round(int a, int b, int sft_amount)
{
int sum = a + b;
switch (sft_amount)
{
case 1:
return sum >> 1;
case 2:
return sum >> 2;
}
return 0;  // Should never be reached
}
```

You can also use these techniques to avoid creating a multiplier or divider. This technique is particularly beneficial for division operations because the hardware responsible for the division is large and relatively slow.

Leave Loops Rolled Up

Sometimes developers unroll loops to achieve better results using a C compiler. Because the C2H Compiler attempts to pipeline all loops, unrolling loops is unnecessary for C2H code. In fact, unrolled loops tend to produce worse results because the C2H Compiler creates extra logic. It is best to leave the loop rolled up. Example 5–30 shows an accumulator algorithm that was unrolled in order to execute faster on a processor.

Example 5–30. Unrolled Loop

```
int sum(int* arr)
{
int i;
int result = 0;
for (i = 0; i < 100; i += 4)
{
result += *arr++;
result += *arr++;
result += *arr++;
result += *arr++;
}
return result;
}
```

This function is passed an array of 100 integers, accumulates each element, and returns the sum. To achieve higher performance on a processor, the developer has unrolled the inner loop four times, reducing the loop overhead by a factor of four when executed on a processor. When the C2H Compiler maps this code to hardware, there is no loop overhead because the accelerator executes the loop overhead statements concurrently with the loop body.

As a result of unrolling the code, the C2H Compiler creates four times more logic because four separate assignments are used. The C2H Compiler creates four Avalon-MM master ports in the loop. However, an Avalon-MM master port can only perform one read or write operation at any given time. The four master ports must interleave their accesses, eliminating any advantage of having multiple masters.

Example 5–30 shows how resource sharing (memory) can cause parallelism to be nullified. Instead of using four assignments, roll this loop up as shown in Example 5–31.

Example 5–31. Rolled-Up Loop

```
int sum(int* arr)
{
int i;
int result = 0;
for (i = 0; i < 100; i++)
{
result += *arr++;
}
return result;
}
```

This implementation achieves the same throughput as the previous unrolled example because this loop can potentially iterate every clock cycle. The unrolled algorithm iterates every four clock cycles due to memory stalls. Because these two algorithms achieve the same throughput, the added benefit of the rolling optimization is savings on logic resources such as Avalon-MM master ports and additional accumulation logic.

Use ++ to Sequentially Access Arrays

The unrolled version of the `sum` function in Example 5–30 uses `*arr++` to sequentially access all elements of the array. This procedure is more efficient than the alterative shown in Example 5–32.

Example 5–32. Traversing Array with Index

```
int sum(int* arr)
{
int i;
int result = 0;
for (i = 0; i < 100; i++)
{
result += arr[i];
}
return result;
}
```

The C2H Compiler must create pointer dereferences for both `arr[i]` and `*arr++`. However, the instantiated logic is different for each case. For `*arr++` the value used to address memory is the pointer value itself, which is capable of incrementing. For `arr[i]` the accelerator must add base address `arr` to the counter value `i`. Both require counters, however in the case of `arr[i]` an adder block is necessary, which creates more logic.

Avoid Excessive Pointer Dereferences

Any pointer dereference via the dereference operator `*` or array indexing might create an Avalon-MM master port. Avoid using excessive pointer dereference operations because they lead to both additional logic which degrades the f_{MAX} of the design.

Any local arrays within the accelerated function instantiate on-chip memory resources. Do not declare large local arrays because the amount of on-chip memory is limited and excessive use affects the routing of the design.

Avoid Multipliers

Embedded multipliers have become a standard feature of FPGAs; however, they are still limited resources. When you accelerate source code that uses a multiplication function, the C2H accelerator instantiates a multiplier. Embedded multiplier blocks have various modes that allow them to be segmented into smaller multiplication units depending on the width of the data being used. They also have the ability to perform multiply and accumulate functionality.

When using multipliers in accelerated code validate the data width of the multiplication to reduce the logic. The embedded multiplier blocks handle 9 by 9 (`char *char`), 18 by 18 (`short *short`), and 36 by 36 (`long *long`) modes which are set depending on the size of the largest width input. Reducing the input width of multiplications not only saves resources, but also improves the routing of the design, because multiplier blocks are fixed resources. If multiplier blocks are not available or the design requires too many multiplications, the Quartus II software uses logic elements to create the multiplication hardware. Avoid this situation if possible, because multipliers implemented in logic elements are expensive in terms of resources and design speed.

Multiplications by powers of two do not instantiate multiplier logic because the accelerator can implement them with left shift operations. The C2H Compiler performs this optimization automatically, so it is not necessary to use the `<<` operator. When multiplication is necessary, try to use powers of two in order to save logic resources and to benefit from the fast logic created for this operation. An assignment that uses a multiplication by a power of two becomes a register-to-register path in which the data is shifted in the system interconnect fabric.

When multiplying by a constant the Quartus II software optimizes the LEs either using memory or logic optimizations. Example 5–33 shows an optimization for multiplication by a constant.

Example 5–33. Multiplication by Constants

```
/* This multiplication by a constant is optimized */
y = a × 3;

/*The optimization is shift and add: (2*a + a = 3*a) */
y = a << 1 + a
```

C2H offloads the intelligence of multiplies, divides, and modulo to Quartus II synthesis to do the right thing when possible.

Avoid Arbitrary Division

If at all possible, avoid using arbitrary division in accelerated functions, including the modulus `%` operator. Arbitrary division occurs whenever the divisor is unknown at compile time. True division operations in hardware are expensive and slow.

Example 5–34. Arbitrary Division (Expensive, Slow):

```
z = y / x;  /* x can equal any value */
```

The exception to this rule is division by denominators which are positive powers of two. Divisions by positive powers of two simply become binary right-shift operations. Dividing by two can be accomplished by shifting the value right one bit. Dividing by four is done by shifting right two bits, and so on. If the accelerated function uses the `/` division operator, and the right-hand argument is a constant power of two, the C2H Compiler converts the divide into a fixed-bit shift operation. In hardware, fixed-bit shift operations result in only wires, which are free.

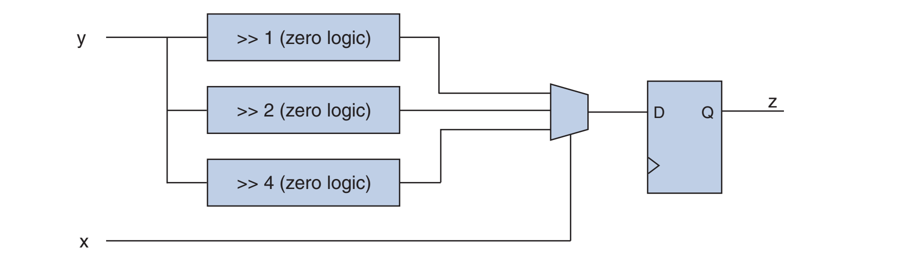
If a division operation in an accelerated function always uses a denominator that is a power of two, but can use various multiples of two, you can use a ternary operation to convert the divides to the appropriate fixed-bit shift, as shown in Example 5–35.

Example 5–35. Division using Shifts with a Ternary Operator (Cheap, Fast)

```
z = (x == 2)? y >> 1:((x == 4)? y >> 2: y >> 4);
```

Figure 5–8 shows the hardware generated for Example.

Figure 5–8. . Ternary Shift Divide



The logic created by this optimization is relatively cheap and fast, consisting of a multiplexer and minimal control logic. Because the assignments to z are just shifted copies of y the multiplexer is the only logic in the register-to-register path. If there are many possible denominator values, explore the tradeoff between latency and frequency discussed in the"Improve Conditional Frequency" on page 5–37.

The other possible optimization to avoid generating an expensive and slow division circuit is to implement a serial divider. Serial dividers have a high latency, but tend not to degrade f_{MAX}. Another benefit of using serial division is the relatively low cost of the hardware generated because the operations performed are on bits instead of words.

You can use macros in **c2h_division.h** and **c2h_modulo.h** to implement serial division or modulo operations in your own system. These files are available on the Nios II literature page. A hyperlink to the software files appears next to *Optimizing Nios II C2H Compiler Results* (this document), at **www.altera.com/literature/lit-nio2.jsp.** The two header files are distributed in a zip file.

Use Masks

Both the C compiler for a 32-bit processor and the C2H Compiler convert data types smaller than integers to 32-bit integers. If you want to override this default behavior to save logic and avoid degrading the f_{MAX} of the design, add a bitwise `AND` with a mask. In Example 5–36, the C2H Compiler promotes `b1` and `b2` to 32-bit integers when performing the addition so that it instantiates a 32-bit adder in hardware. However, because `b1` and `b2` are unsigned characters, the sum of `b1` and `b2` is guaranteed to fit in nine bits, so you can mask the addition to save bits. The C2H Compiler still instantiates a 32-bit adder but Quartus II synthesis removes the unnecessary bits, resulting in a 9-bit adder in hardware.

Example 5–36. Use Bitwise And with a Mask

```
unsigned int add_chars(unsigned char b1, unsigned char b2)
{
  return (b1 + b2) & 0x1ff;
}
```

This optimization can cause a failure if you mistakenly reduce the width of the calculation so that needed data resolution is lost. Another common mistake is to use bit masks with signed data. Signed data, stored using 2's complement format, requires that the accelerator preserve and extend the sign bit through the masking operation.

Use Powers of Two in Multi-Dimensional Arrays

A conventional C compiler implements a multidimensional array as a one-dimensional array stored in row-major order. For example, a two-dimensional array might appear as follows:

```
#define NUM_ROWS 10
#define NUM_COLS 12
int arr2d[NUM_ROWS][NUM_COLS];
```

The first array index is the row and the second is the column. A conventional C compiler implements this as a one-dimensional array with NUM_ROWS x NUM_COLS elements. The compiled code computes the offset into the one-dimensional array using the following equation:

```
offset = row * NUM_COLS + col;
```

The C2H Compiler follows this implementation of multidimensional arrays. Whenever your C code indexes into a multidimensional array, an implicit multiplication is created for each additional dimension. If the multiplication is by a power of two, the C2H Compiler implements the multiplication with a wired shift, which is free. If you can increase that dimension of the array to a power of two, you save a multiplier. This optimization comes at the cost of some memory, which is cheap. In the example, just make the following change:

```
#define NUM_COLS 16
```

To avoid all multipliers for multidimensional array accesses of *<n>* dimensions, you must use an integer power of two array size for each of the final *<n-1>* dimensions. The first dimension can have any length because it does not influence the decision made by the C2H Compiler to instantiate multipliers to create the index.

Use Narrow Local Variables

The use of local variables that are larger data types than necessary can waste hardware resources in an accelerator. Example 5–37 includes a variable that is known to contain only the values 0–229. Using a `long int` variable type for this variable creates a variable that is much larger than needed. This type of optimization is usually not applicable to pointer variables. Pointers always cost 32 bits, regardless of their type. Reducing the type size of a pointer variable affects the size of the data the pointer points to, not the pointer itself. It is generally best to use large pointer types to take advantage of wide memory accesses. Refer to "Use Wide Memory Accesses" on page 5–16 for details.

Example 5–37. Wide Local Variable i Costs 32 Bits

```
int i;
int var;
for(i = 0; i < 230; i++)
{
var += *ptr + i;
}
```

An `unsigned char` variable type, as shown in Example 5–38, is large enough because it can store values up to 255, and only costs 8 bits of logic, whereas a `long int` type costs 32 bits of logic. Excessive logic utilization wastes FPGA resources and can degrade system f_{MAX}.

Example 5–38. Narrow Local Variable i Costs 8 Bits

```
unsigned char i;
int var;
for(i = 0; i < 230; i++)
{
var += *ptr + i;
}
```

Optimizing Memory Connections

The following sections discuss ways to optimize memory connectivity.

Remove Unnecessary Connections to Memory Slave ports

The Avalon-MM master ports associated with the `src` and `dst` pointers in Example 5–39 are connected to all of the Avalon-MM slave ports that are connected to the processor's data master. Typically, the accelerator does not need to access all these slave ports. This extra connectivity adds unnecessary logic to the system interconnect fabric, which increases the hardware resources and potentially creates long timing paths, degrading f_{MAX}.

The C2H Compiler supports pragmas added to your C code to inform the C2H Compiler which slave ports each pointer accesses in your accelerator. For example, if the `src` and `dst` pointers can only access the DRAM (assume it is called **dram_0**), add these pragmas before `memcpy` in your C code.

```
#pragma altera_accelerate connect_variable memcpy/dst to dram_0
#pragma altera_accelerate connect_variable memcpy/src to dram_0
```

Example 5–39. Memory Interconnect

```
void memcpy(char* dst, char* src, int num_bytes)
{
while (num_bytes-- > 0)
{
*dst++ = *src++;
}
}
```

These pragmas state that `dst` and `src` only access the **dram_0** component. The C2H Compiler connects the associated Avalon-MM ports only to the **dram_0** component.

Reduce Avalon-MM Interconnect Using #pragma

Accelerated functions use Avalon-MM ports to access data related to pointers in the C code. By default, each master generated connects to every memory slave port that is connected to the Nios II data master port. This connectivity can result in large amounts of arbitration logic when you generate an SOPC Builder system, which is expensive and can degrade system f_{MAX}. In most cases, pointers do not need to access every memory in the system.

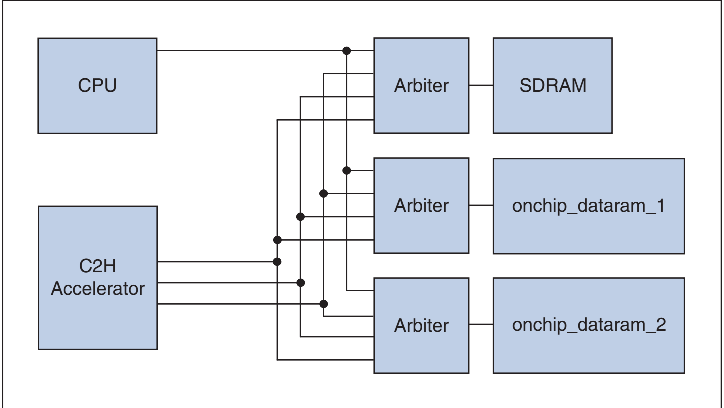
You can reduce the number of master-slave port connections in your SOPC Builder system by explicitly specifying the memories to which a pointer dereference must connect. You can make connections between pointers and memories with the `connect_variable` pragma directive, as shown in Example 5–40. In Figure 5–9, three pointers, `output_data`, `input_data1`, and `input_data2` are connected to

memories named `sdram`, `onchip_dataram1`, and `onchip_dataram2`, respectively. Using the `connect_variable` pragma directive ensures that each of the accelerated function's three Avalon-MM master ports connects to a single memory slave port. The result is a more efficient overall because it has no unnecessary master-slave port connections.

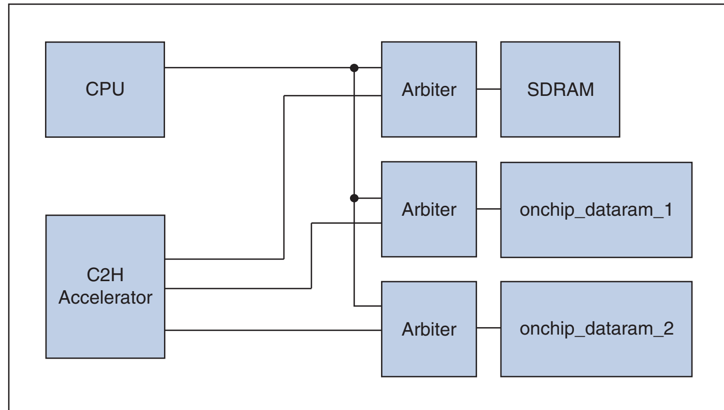
Example 5–40. Reducing Memory Interconnect

```
#pragma altera_accelerate connect_variable my_c2h_function/output_data to sdram

#pragma altera_accelerate connect_variable my_c2h_function/input_data1 to onchip_dataram1
#pragma altera_accelerate connect_variable my_c2h_function/input_data2 to onchip_dataram2

void my_c2h_function( int *input_data1,
                      int *input_data2,
                      int* output_data )
{
  char i;
  for( i = 0; i < 52;  i++ )
  {
    *(output_data + i) = *(input_data1 + i) + *(input_data2 + i);
  }
}
```

Figure 5–9. Pragma Connect



All connections (unnecessary arbitration logic)



Only necessary connections (less arbitration logic)

Remove Unnecessary Memory Connections to Nios II Processor

As part of your optimization, you might have added on-chip memories to the system to allow an accelerated function access to multiple pointers in parallel, as in “Segment the Memory Architecture” on page 5–18. During implementation and debug, it is important that these on-chip memories have connections to both the appropriate accelerator Avalon-MM master port and to the Nios II data master port, so the function can run in both accelerated and non-accelerated modes. In some cases however, after you are done debugging, you can remove the memory connections to the Nios II data master if the processor does not access the memory when the function is accelerated. Removing connections lowers the cost and avoids degrading system f_{MAX}.

Optimizing Frequency Versus Latency

The following sections describe tradeoffs you can make between frequency and latency to improve performance.

Improve Conditional Latency

Algorithms that contain `if` or `case` statements use registered control paths when accelerated. The C2H Compiler accelerates the code show in Example 5–41 in this way.

Example 5–41. Registered Control Path

```
if(testValue < Threshold)
{
a = x;
}
else
{
a = y;
}
```

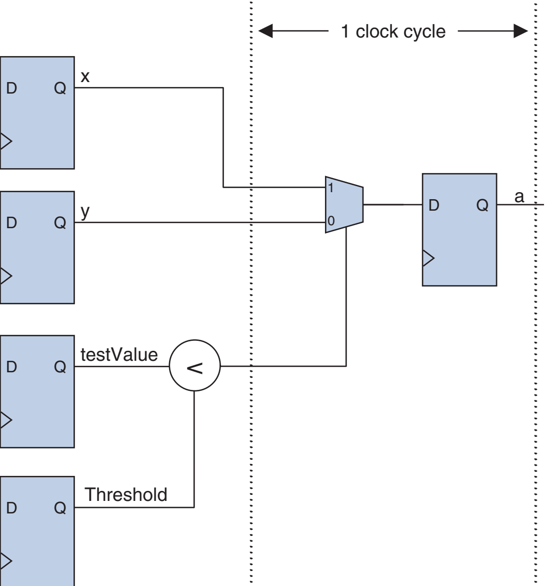
You can modify your software to make use of the ternary operator, (`?:`), as in Example 5–42, to reduce the latency of the control path. The ternary operator does not register signals on the control path, so this optimization results in lower latency at the expense of f_{MAX}. This optimization primarily helps reduce the CPLI of the accelerator when a data dependency prevents the conditional statement from becoming fully pipelined. Do not use this optimization if the CPLI of the loop containing the conditional statement is already equal to one.

Example 5–42. Unregistered Control Path

```
a = (testValue < Threshold)? x : y;
```

Figure 5–10 shows the hardware generated for Example 5–42.

Figure 5–10. Conditional Latency Improvement



Improve Conditional Frequency

If you wish to avoid degrading f_{MAX} in exchange for an increase in latency, consider removing ternary operators. By using an `if` or `case` statement to replace the ternary operator the control path of the condition becomes registered and shortens the timing paths in that portion of the accelerator. In the case of the conditional statement being executed infrequently (outside of a loop), this optimization might prove a small price to pay to increase the overall frequency of the hardware design.

Example 5–43 and Example 5–44 show how you can rewrite a ternary operator as an `if` statement.

Example 5–43. Unregistered Conditional Statement

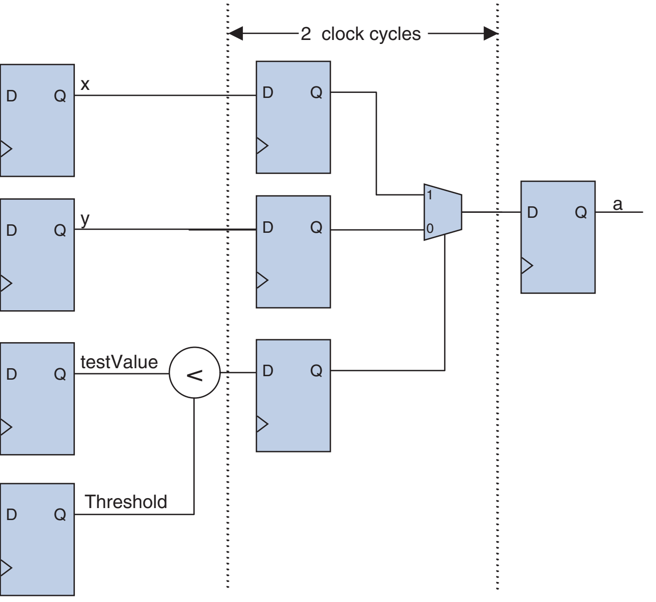
```
a = (testValue < Threshold)? x : y;
```

Example 5–44. Registered Conditional Statement

```
if(testValue < Threshold)
{
a = x;
}
else
{
a = y;
}
```

Figure 5–11 shows the hardware the C2H Compiler generates for Example 5–44.

Figure 5–11. Conditional Frequency Improvement



Improve Throughput

To increase the computational throughput, focus on two main areas: achieving a low CPLI, and performing many operations within one loop iteration.

Avoid Short Nested Loops

Because a loop has a fixed latency before any iteration can occur, nesting looping structures can lead to unnecessary delays. The accelerator incurs the loop latency penalty each time it enters the loop. Rolling software into loops adds the possible benefit of pipelining, and the benefits of this pipelining usually outweigh the latency associated with loop structures. Generally, if the latency is greater than the maximum number of iterations times the CPLI then the looping implementation is slower. You must take into account that leaving a loop unrolled usually increases the resource usage of the hardware accelerator.

Example 5–45. Nested Loops

```
for(loop1 = 0; loop1 < 10; loop1++) /* Latency = 3, CPLI = 2 */
{
  /* statements requiring two clock cycles per loop1 iteration */
  for(loop2 = 0; loop2 < 5; loop2++)  /* Latency = 10, CPLI = 1 */
  {
     /* statements requiring one clock cycle per loop2 iteration */
  }
}
```

Assuming no memory stalls occur, the total number of clock cycles is as follows:

$$Innerloop = latency + (iterations - 1)(CPLI + innerlooptime)$$

$$Innerloop = 10 + 4(1 + 0)$$

$$Innterloop = 14 cycles$$

$$Outerloop = 3 + 9(2 + 14)$$

$$Outerloop = 147 cycles$$

Due to the high latency of the inner loop the total time for this example is 147 clock cycles.

Example 5–46. Single Loop

```
for(loop1 = 0; loop1 < 10; loop1++) /* Latency = 3, CPLI = 7 */
{
  /* statements requiring two clock cycles per loop1 iteration */
  /* statements that were previously contained in loop2 */
}
```

Assuming no memory stalls occur, the total number of clock cycles is as follows:

$$Outerloop = latency + (iterations - 1)(CPLI + innerlooptime)$$

$$Outerloop = 3 + 9(7 + 0)$$

$$Outerloop = 66 cycles$$

The inner loop (`loop2`) has been eliminated and consequently is 0 in these equations. Combining the inner loop with the outer loop dramatically decreases the total time to complete the same outer loop. This optimization assumes that unrolling the inner loop resulted in adding five cycles per iteration to the outer loop. The combination of the loops would most likely result in a hardware utilization increase which you must take into consideration.

Remove In-place Calculations

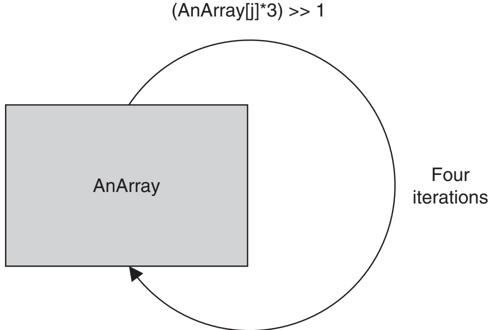
Some software algorithms perform in-place calculations, in which results overwrite the input data as they are calculated. This technique conserves memory, but produces suboptimal performance when compiled to a C2H accelerator. Example 5–47 shows such an algorithm. Unfortunately this approach leads to memory stalls because in-place algorithms read and write to the same memory locations.

Example 5–47. Two Avalon-MM Ports Using The Same Memory

```
for(i = 0; i < 4; i++)
{
for(j = 0; j < 1024; j++)
{
AnArray[j] = (AnArray[j] * 3) >> 1;
}
}
```

Figure 5–12 shows the dataflow in hardware generated for Example.

Figure 5–12. In-Place Calculation



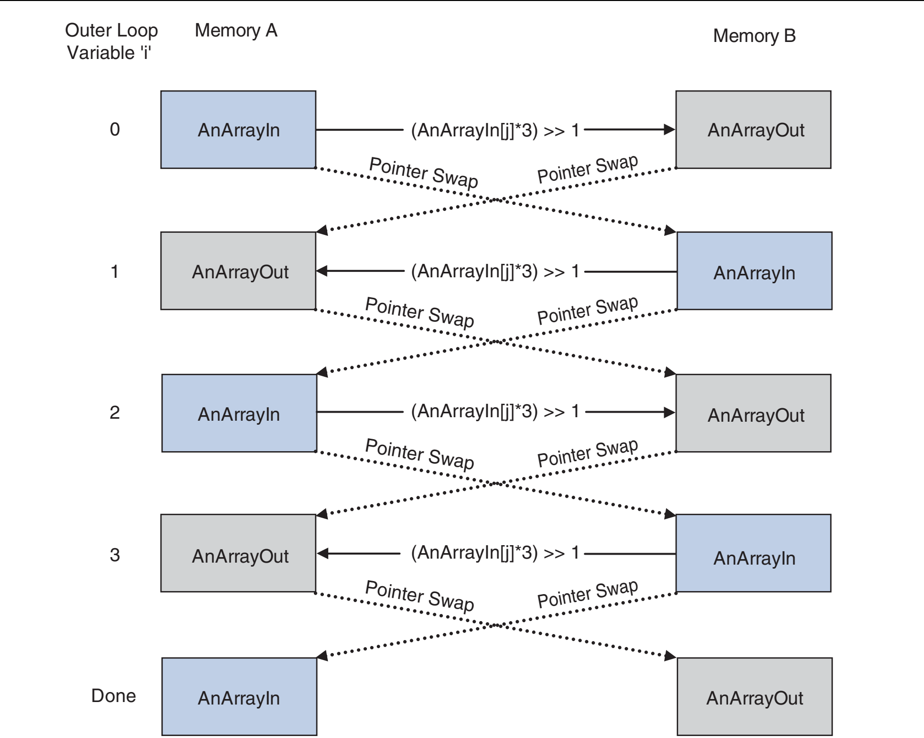
To solve this problem, remove the in-place behavior of the algorithm by adding a "shadow" memory to the system, as shown in Example 5–48. Instead of the input and output residing in the same memory, each uses an independent memory. This optimization prevents memory stalls because the input and output data reside in separate memories.

Example 5–48. Two Avalon-MM Ports Using Separate Memories

```
int * ptr;
for(i = 0; i < 4; i++)
{
for(j = 0; j < 1024; j++)
{
/* In from one memory and out to the other */
AnArrayOut[j] = (AnArrayIn[j] * 3) >> 1;
}
/* Swap the input and output pointers and do it all
over again */
ptr = AnArrayOut;
AnArrayOut = AnArrayIn;
AnArrayIn = ptr;
}
```

Figure 5–13 shows the dataflow of hardware generated for Example 5–48.

Figure 5–13. In-Place Calculation



You can also use this optimization if the data resides in on-chip memory. Most on-chip memory can be dual-ported to allow for simultaneous read and write access. With a dual-port memory, the accelerator can read the data from one port without waiting for the other port to be written. When you use this optimization, the read and write addresses must not overlap, because that could lead to data corruption. A method for preventing a read and a write from occurring simultaneously at the same address is to read the data into a variable before the write occurs.

Replace Arrays

Often software uses data structures that are accessed via a base pointer location and offsets from that location, as shown in Example 5–49. When the hardware accelerator accesses the data in these structures, memory accesses result.

Example 5–49. Individual Memory Accesses

```
int a = Array[0];
int b = Array[1];
int c = Array[2];
int d = Array[3];
```

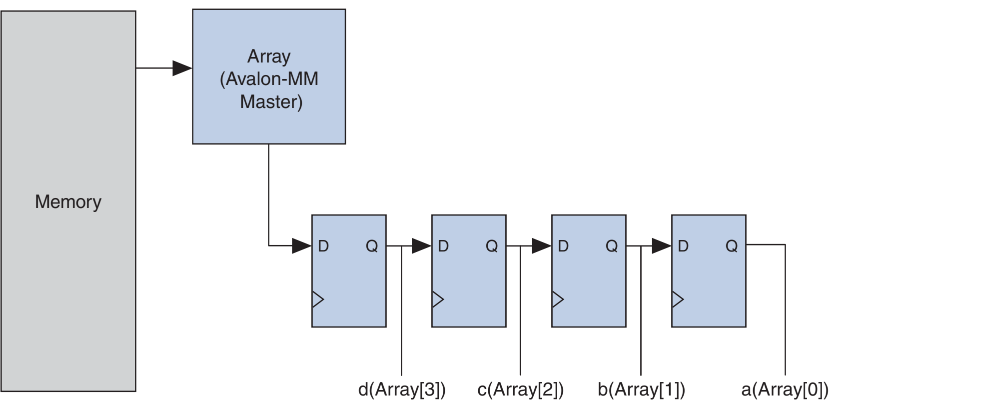
You can replace these memory accesses using a single pointer and registers, as in Example 5–50. The overall structure of the hardware created resembles a FIFO.

Example 5–50. FIFO Memory Accesses

```
/* initialize variables */
int a = 0;
int b = 0;
int c = 0;
int d = 0;
for(i = 0; i < 4; i++)
{
  d = Array[i];
  c = d;
b = c;
  a = b;

}
```

Figure 5–14 shows the hardware generated for Example 5–50.

Figure 5–14. Array Replacement



Use Polled Accelerators

When you create a hardware accelerator using the C2H Compiler, it creates a wrapper file that is linked at compile time, allowing the main program to call both the software and hardware versions of the algorithm using the same function name. The wrapper file performs the following three tasks:

- Writes the passed parameters to the accelerator
- Polls the accelerator to determine when the computation is complete
- Sends the return value back to the caller

Because the wrapper file is responsible for determining the status of the accelerator, the Nios II processor must wait for the wrapper code to complete. This behavior is called blocking.

The hardware accelerator blocks the Nios II processor from progressing until the accelerator has reached completion. The wrapper file is responsible for this blocking action. Using the same pragma statement to create the interrupt include file, you can access the macros defined in it to implement a custom polling algorithm common in systems that do not use a real time operating system.

Instead of using the interrupt to alert Nios II that the accelerator has completed its calculation, the software polls the `busy` value associated with the accelerator. The macros necessary to manually poll the accelerator to determine if it has completed are in the include file created under either the **Debug** or **Release** directory of your application project. These macros are shown in Table 5–5.

Table 5–5. C2H Accelerator Polling Macros

Purpose	Macro Name
Busy value	`ACCELERATOR_`*<Project Name>*`_`*<Function Name>*`_BUSY()`
Return value	`ACCELERATOR_`*<Project Name>*`_`*<Function Name>*`_GET_RETURN_VALUE()`

While the accelerator is busy, the rest of the software must not attempt to read the return value because it might be invalid.

© June 2008 Altera Corporation

Use an Interrupt-Based Accelerator

The blocking behavior of a polled accelerator might be undesirable if there are processing tasks which the Nios II processor can carry out while the accelerator is running. In this case, you can create an interrupt-based accelerator.

Create the hardware accelerator with the standard flow first, because interrupts add an extra level of complexity. Before proceeding to the interrupt flow, debug the system to make sure the accelerator behaves correctly. Add enhancements in polled mode, as well.

To use the hardware accelerator in a non-blocking mode, add the following line to your function source code:

```
#pragma altera_accelerate enable_interrupt_for_function<function name>
```

At the next software compilation, the C2H Compiler creates a new wrapper file containing all the macros needed to use the accelerator and service the interrupts it generates. The hardware accelerator does not have an IRQ level so you must open the system in SOPC Builder and manually assign this value. After assigning the IRQ level you must click the **Generate** button to regenerate your SOPC Builder system.

The macros necessary to service the accelerator interrupt are in the **include** file created under either the **Debug** or **Release** directory of your application project. These macros are shown in Table 5–6.

Table 5–6. C2H Accelerator Interrupt Macros

Purpose	Macro Name
IRQ level value	ACCELERATOR_<*Project Name*>_<*Function Name*>_IRQ()
Return value	ACCELERATOR_<*Project Name*>_<*Function Name*>_GET_RETURN_VALUE()
Interrupt clear	ACCELERATOR_<*Project Name*>_<*Function Name*>_CLEAR_IRQ()

Refer to the *Exception Handling* chapter in the *Nios II Software Developer's Handbook* for more information about creating interrupt service routines.

Glossary

This document uses the following terminology:

- **Accelerator throughput—**the throughput achieved by a C2H accelerator during a single invocation. Accelerator throughput might be less than peak throughput if pipeline stalls occur. Accelerator throughput does not include latency. See also CPLI, throughput, peak throughput, application throughput.
- **Application throughput—**the throughput achieved by a C2H accelerator in the context of the application, involving multiple accelerator invocations and including the number of cycles of latency.
- **Barrel shifter –** hardware that shifts a byte or word of data an arbitrary number of bits in one clock cycle. Barrel shifters are fast and expensive, and can degrade f_{MAX}.

- **Cache coherency**—the integrity of cached data. When a processor accesses memory through a cache and also shares that memory with a coprocessor (such as a C2H accelerator), it must ensure that the data in memory matches the data in cache whenever the coprocessor accesses the data. If the coprocessor can access data in memory that has not been updated from the cache, there is a cache-coherency problem.
- **Compute-limited**—describes algorithms whose speed is restricted by how fast data can be processed. When an algorithm is compute-limited, there is no benefit from increasing the efficiency of memory or other hardware. See also data-limited.
- **Control path**—a chain of logic controlling the output of a multiplexer
- **CPLI**—cycles per loop iteration. The number of clock cycles required to execute one loop iteration. CPLI does not include latency.
- **Critical timing path**—the longest timing path in a clock domain. The critical timing path limits f_{MAX} or the entire clock domain. See also timing path.
- **Data dependency**—a situation where the result of an assignment depends on the result of one or more other assignments, as in Example 5–5.
- **Data-limited**—describes algorithms whose speed is restricted by how fast data can be transferred to or from memory or other hardware. When an algorithm is data-limited, there is no benefit from increasing processing power. See also compute-limited.
- **DRAM**—dynamic random access memory. It is most efficient to access DRAM sequentially, because there is a time penalty when it is accessed randomly. SDRAM is a common type of DRAM.
- **Latency**—a time penalty incurred each time the accelerator enters a loop.
- **Long timing path**—a critical timing path that degrades f_{MAX}.
- **Peak throughput**—the throughput achieved by a C2H accelerator, assuming no pipeline stalls and disregarding latency. For a given loop, peak throughput is inversely proportional to CPLI. See also throughput, accelerator throughput, application throughput, CPLI, latency.
- **Rolled-up loop**—A normal C loop, implementing one algorithmic iteration per processor iteration. See also unrolled loop.
- **SDRAM**—synchronous dynamic random access memory. See DRAM.
- **SRAM**—static random access memory. SRAM can be accessed randomly without a timing penalty.
- **Subfunction**—a function called by an accelerated function. If `apple()` is a function, and `apple()` calls `orange()`, `orange()` is a subfunction of `apple()`. If `orange()` calls `banana()`, `banana()` is also a subfunction of `apple()`.
- **Throughput**—the amount of data processed per unit time. See also accelerator throughput, application throughput, and peak throughput.
- **Timing path**—a chain of logic connecting the output of a hardware register to the input of the next hardware register.

- **Unrolled loop**—A C loop that is deconstructed to implement more than one algorithmic iteration per loop iteration, as illustrated in Example 5–30. See also rolled-up loop.

Referenced Documents

This chapter references the following documents:

- *AN391: Profiling Nios II Systems*
- *Cache and Tightly-Coupled Memory* in the *Nios II Processor Reference Handbook*
- *Exception Handling* chapter in the *Nios II Software Developer's Handbook*
- *Nios II C2H Compiler User Guide*
- *Nios II Hardware Development Tutorial*
- *Nios II Processor Reference Handbook*

Document Revision History

Table 5–7 shows the revision history for this chapter.

Table 5–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release	This chapter was previously released as *AN 420: Optimizing Nios II C2H Compiler Results*.

Section III. System-Level Design

This section of the Embedded Design Handbook recommends design styles and practices for developing, verifying, debugging, and optimizing hardware for use in Altera FPGAs. The section introduces concepts to new users of Altera's devices and helps to increase the design efficiency of the experienced user.

This section includes the following chapters:

- Chapter 6, Avalon Memory-Mapped Design Optimizations
- Chapter 7, Memory System Design
- Chapter 8, Hardware Acceleration and Coprocessing
- Chapter 9, Verification and Board Bring-Up
- Chapter 10, Interfacing an External Processor to an Altera FPGA

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

6. Avalon Memory-Mapped Design Optimizations

ED51007-1.1

The Avalon® Memory-Mapped (Avalon-MM) system interconnect fabric is a flexible, partial crossbar fabric that connects master and slave components. Understanding and optimizing this system interconnect fabric in can help you create higher performance designs. When you use the Altera® system-on-a-programmable-chip (SOPC) design tool, SOPC builder automatically generates optimized interconnect logic to your specifications, saving you from time-consuming and error-prone task.

This chapter provides recommendations to optimize the performance, resource utilization, and power consumption of your Avalon-MM design. The following topics are discussed:

- Selecting Hardware Architecture
- Understanding Concurrency
- Increasing Transfer Throughput
- Increasing System Frequency
- Reducing Logic Utilization
- Reducing Power Utilization

One of the key advantages of FPGAs for system design is the high availability of parallel resources. SOPC Builder uses the parallel resources inherent in the FPGA fabric to maximize concurrency. You use the SOPC Builder GUI to specify the connectivity between blocks of your design. SOPC Builder automatically generates the optimal HDL from your specification.

Selecting Hardware Architecture

Hardware systems typically use one of four architectures to connect the blocks of a design:

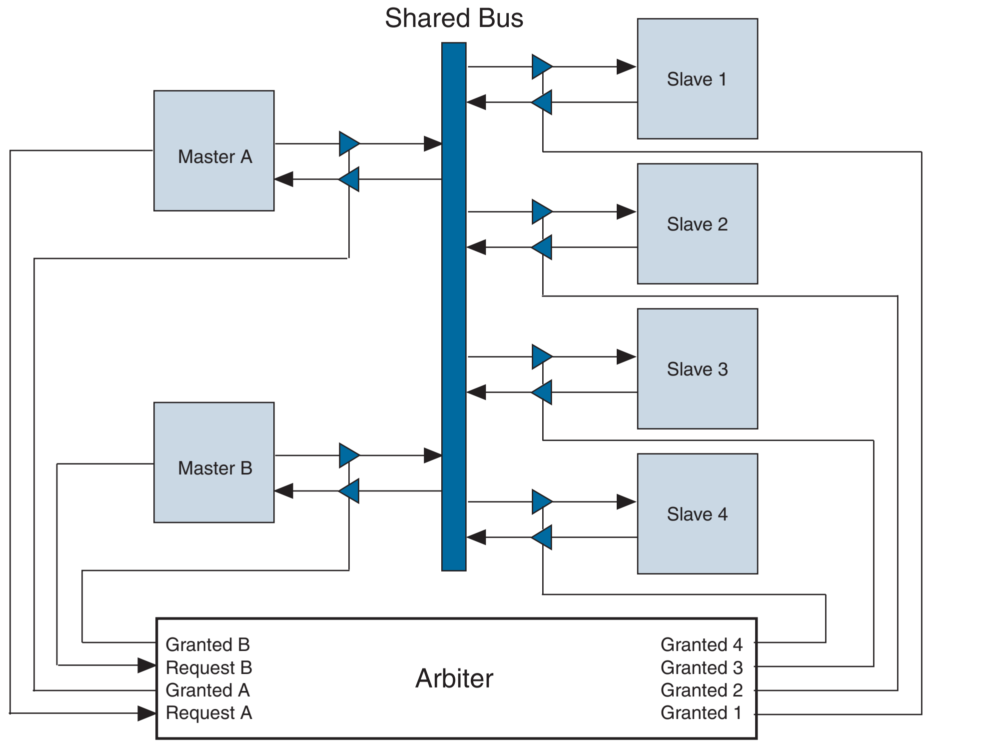
- Bus
- Full Crossbar Switch
- Partial Crossbar Switch
- Streaming

No single architecture can be used efficiently for all systems. The following sections discuss the characteristics, advantages and disadvantages of each of these interconnect architectures.

Bus

Bus architectures can achieve relatively high clock frequencies at the expense of little or no concurrency. Bus architectures connect masters and slaves using a common arbitration unit. The arbiter must grant a master access to the bus before a data transfer can occur. A shared bus architecture can lead to a significant performance penalty in systems with many bus masters because all masters compete for access to the shared bus rather than a particular slave device.

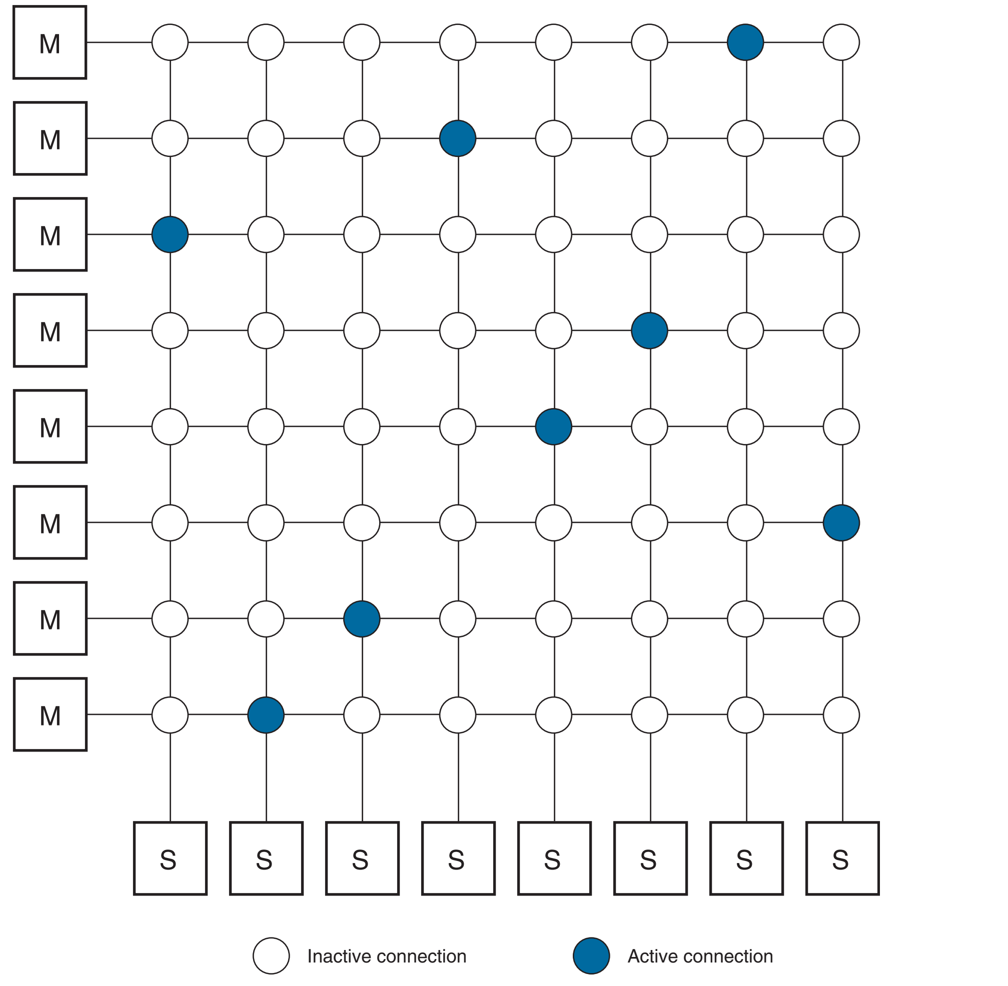
Figure 6–1. Bus Architecture



Full Crossbar Switch

Crossbar switches, unlike bus architectures, support concurrent transactions. A crossbar switch allows any number of masters to connect to any number of slaves. Networking and high performance computing applications typically use crossbars because they are flexible and provide high throughput. Crossbars are implemented with large multiplexers. The crossbar switch includes the arbitration function. Crossbars provide a high degree of concurrency. The hardware resource utilization grows exponentially as more masters and slaves are added; consequently, FPGA designs avoid large crossbar switches because logic utilization must be optimized.

Figure 6–2. Crossbar Switch



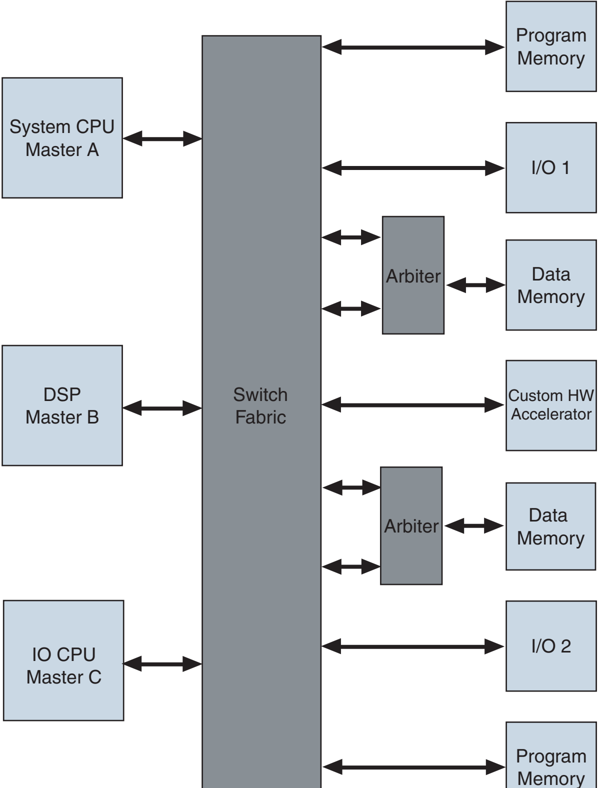
Partial Crossbar Switch

In many embedded systems, individual masters only require connectivity to a subset of the slaves so that a partial crossbar switch provides the optimal connectivity. There are two significant advantages to the partial crossbar switch:

- The reduction in connectivity results in system interconnect fabric that operates at higher clock frequencies
- The system interconnect fabric consumes fewer resources.

These two advantages make partial crossbar switches ideal for ASIC or FPGA interconnect structures. Figure 6–3 illustrates an SOPC Builder system with masters and slaves connected by a partial crossbar switch.

Figure 6–3. Partial Crossbar Switch – SOPC Builder System Interconnect



SOPC Builder generates logic that implements the partial crossbar system interconnect fabric using slave side arbitration. An arbiter included in the system interconnect fabric performs slave side arbitration. This architecture significantly reduces the contention for resources that is inherent in a shared bus architecture. The arbiter selects among all requesting masters so that unless two or more masters are requesting access in the same cycle, there is no contention. In contrast, a shared bus architecture requires all masters to arbitrate for the bus, regardless of the actual slave device to which the masters requires access.

In Figure 6–3, the system CPU has its own program memory and I/O; there is never any contention for these two slave resources. The system CPU shares a memory with the DSP master; consequently, there is a slave-side arbiter to control access. The DSP is the only master that accesses the custom hardware accelerator; there is no contention for this device. The DSP and I/O CPU master share a memory, and access is controlled by a slave-side arbiter. The I/O CPU master has it own program memory and I/O device.

The partial crossbar switch that SOPC Builder generates is ideal in FPGA designs because SOPC Builder only generates the logic necessary to connect masters and slaves that communicate with each other. Using SOPC Builder, you gain the performance of a switch fabric with the flexibility of an automatically generated interconnect architecture. Because SOPC Builder automatically generates the system interconnect fabric, you can regenerate it automatically if your system design changes.

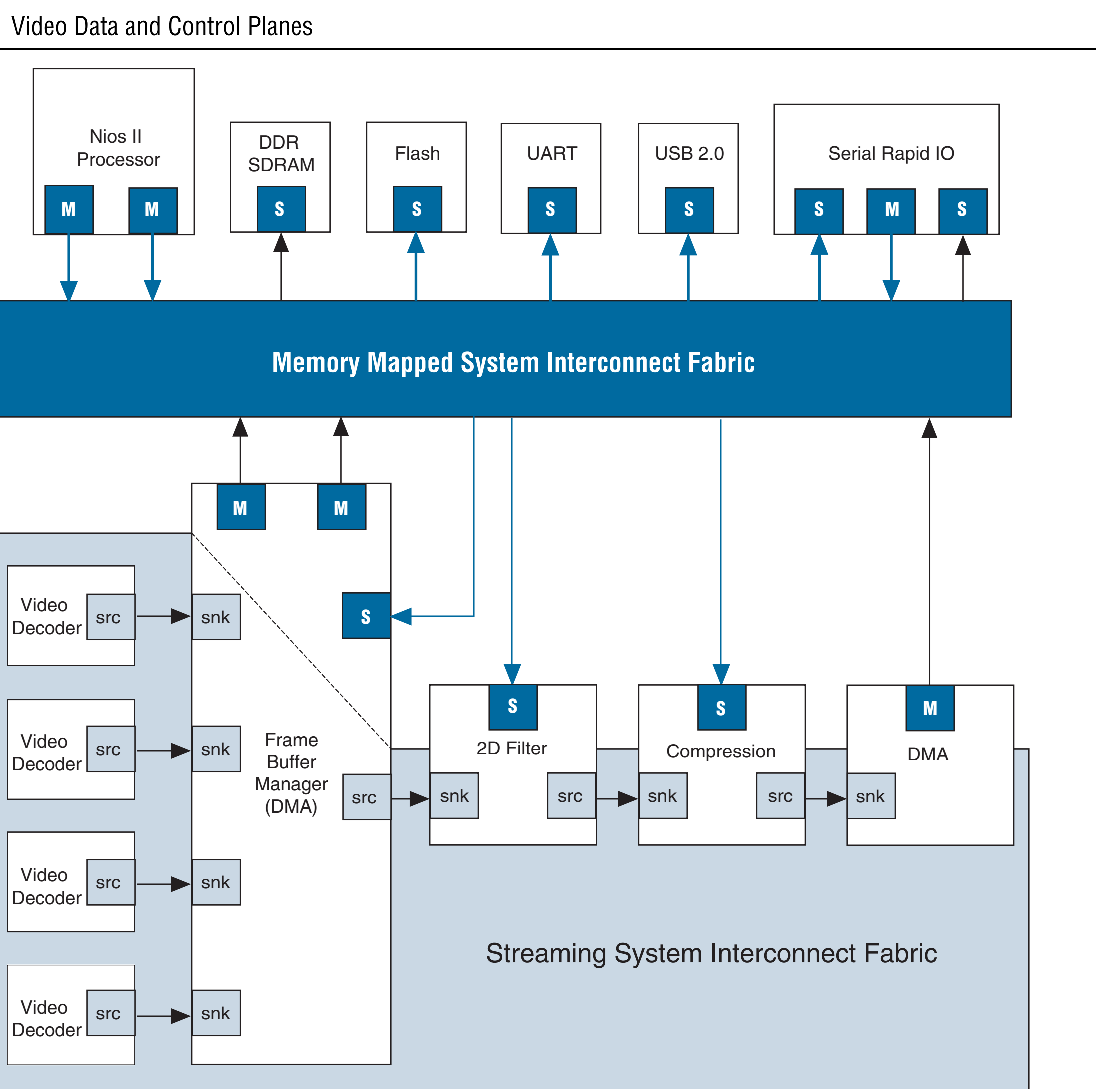
Streaming

Applications that require high speed data transfers use streaming interconnect. SOPC Builder supports Avalon Streaming (Avalon-ST) which creates point-to-point connections between source and sink components. Each streaming connection includes a single source and sink pair, eliminating arbitration. Because SOPC Builder supports both partial crossbar and streaming connections you can design systems that require the partial crossbar for the control plane, typically used to program registers and set up data transfers, and streaming for the data plane, typically used for high speed data transfers.

Full and partial crossbar switches and streaming architectures are all commonly used to implement data planes. The control plane usually includes a processor or state machine to control the flow of data. Full or partial crossbar switches or a shared bus architecture implement control planes.

SOPC Builder generates interconnect logic for both data and control planes. The system interconnect fabric connects Avalon-MM and Avalon-ST interfaces automatically based on connectivity information that you provide. Figure 6–4 shows a video processing application designed using SOPC Builder. This application uses Avalon-MM interfaces for control and Avalon-ST interfaces to transfer data. The video imaging pipeline includes five hardware blocks with Avalon-ST interfaces: a video decoder, a frame buffer, a two-dimensional filter, a compression engine and a direct memory access (DMA) master. All of the hardware blocks, with the exception of the video decoders, also include an Avalon-MM interface for control.

Figure 6–4. Video Data and Control Planes



Dynamic Bus Sizing

A common issue in system design is integration of hardware blocks of different data widths. SOPC Builder automatically adapts mixed width Avalon-MM components by inserting the correct adapter logic between masters and slaves of different widths. For example, if you connect a 32-bit master to a 16-bit slave, the system interconnect fabric creates an adapter that segments the 32-bit transfer into two, separate 16-bit transfers. In creating adapters, the system interconnect fabric employs byte enables to qualify each byte lane.

 © June 2008 Altera Corporation

Understanding Concurrency

One of the key benefits of designing with FPGAs is the re-programmable parallel hardware. Because the underlying FPGA structure supports massive parallelism, the system interconnect fabric is tailored to utilize parallel hardware. You can use parallel hardware to create *concurrency* so that several computational processes are executing at the same time.

The following sections outline other design choices you can make to increase the concurrency in your system.

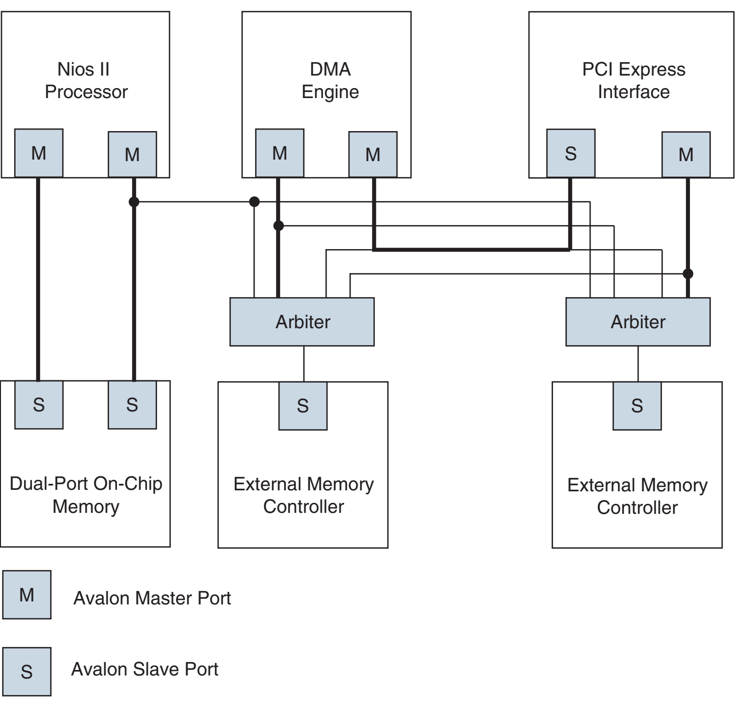
Create Multiple Masters

Your system must have multiple masters to take advantage of the concurrency that the system interconnect fabric supports. Systems that include a Nios® II processor always contain at least two masters, because the Nios II processor includes separate instruction and data masters. Master components typically fall into three main categories:

- General purpose processors, such as Nios II
- DMA engines
- Communication interfaces, such as PCI Express

Because SOPC Builder generates system interconnect fabric with slave side arbitration, every master in your system can issue transfers concurrently. As long as two or more masters are not posting transfers to a single slave, no master stalls. The system interconnect fabric contains the arbitration logic that determines wait states and drives the `waitrequest` signal to the master when a slave must stall. Figure 6–5 illustrates a system with three masters. The bold wires in this figure indicate connections that can be active simultaneously.

Figure 6–5. Multi Master Parallel Access



Create Separate Datapaths

Concurrency is limited by the number of masters sharing any particular slave because the system interconnect fabric uses slave side arbitration. If your system design requires higher data throughput, you can increase the number of masters and slaves to increase the number of transfers that occur simultaneously.

Use DMA Engines

DMA engines also increase data throughput. Because a DMA engine transfers all data between a programmed start and end address without any programmatic intervention, the data throughput is dictated by the components connected to the DMA. The factors that affect data throughput include data width and clock frequency. By including more DMA engines, a system can sustain more concurrent read and write operations as Figure 6–6 illustrates.

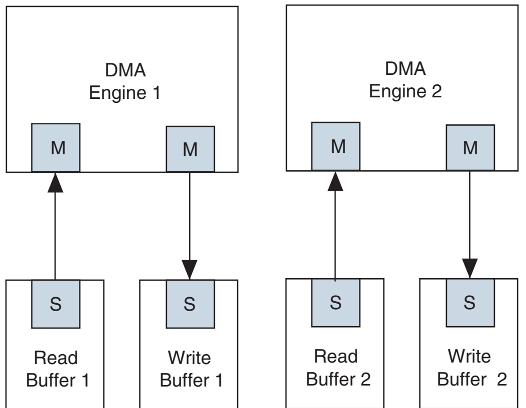
Figure 6–6. Single or Dual DMA Channels

Single DMA Channel

DMA
Engine
M
M
S
Read
Buffer 1
S
Read
Buffer 2
S
Write
Buffer 1
S
Write
Buffer 2

Maximum of one read and one write per clock cycle

Dual DMA Channels

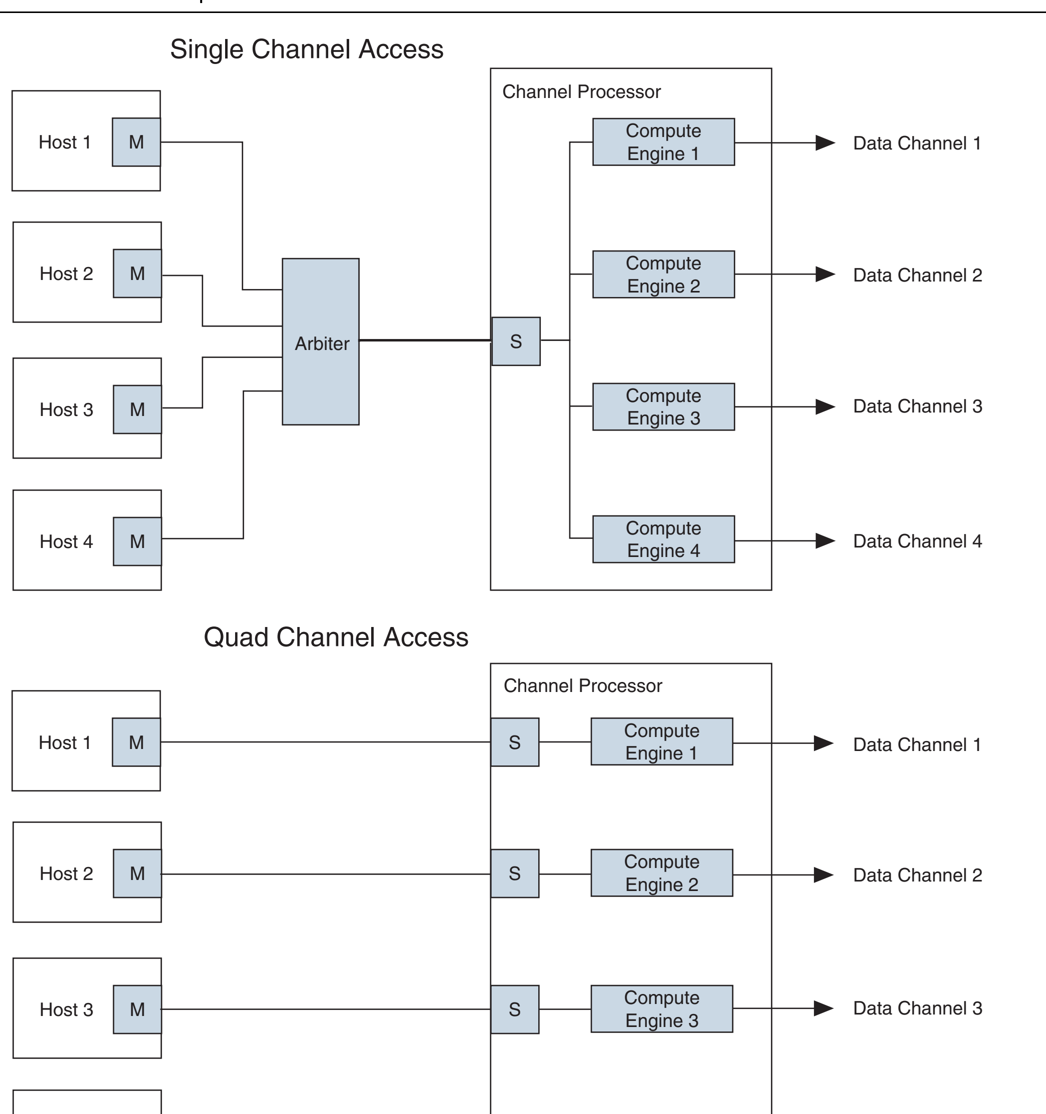


Maximum of two read and two writes per clock cycle

Include Multiple Master or Slave Ports

Creating multiple slave ports for a particular function increases the concurrency in your design. Figure 6–7 illustrates two channel processing

Figure 6–7. Before and After Separate Slaves



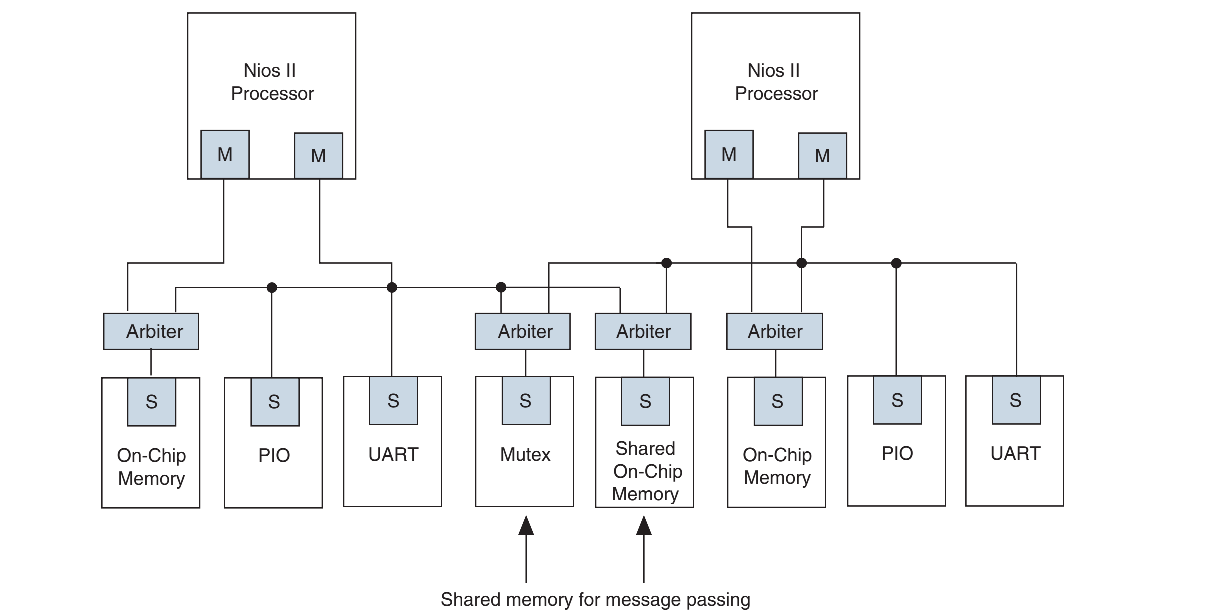
systems. In the first, four hosts must arbitrate for the single slave port of the channel processor. In the second, each host drives a dedicated slave port, allowing all masters to access the component's slave ports simultaneously.

Create Separate Sub-Systems

You can also use hierarchy to sub-divide a system into smaller, more manageable sub-systems. This form of concurrency is implemented by limiting the number of slaves to which a particular master connects. You can create multiple independent sub-systems within a single SOPC Builder system. When communication between sub-systems is necessary, you can use shared memory, message passing, or FIFOs to transfer information.

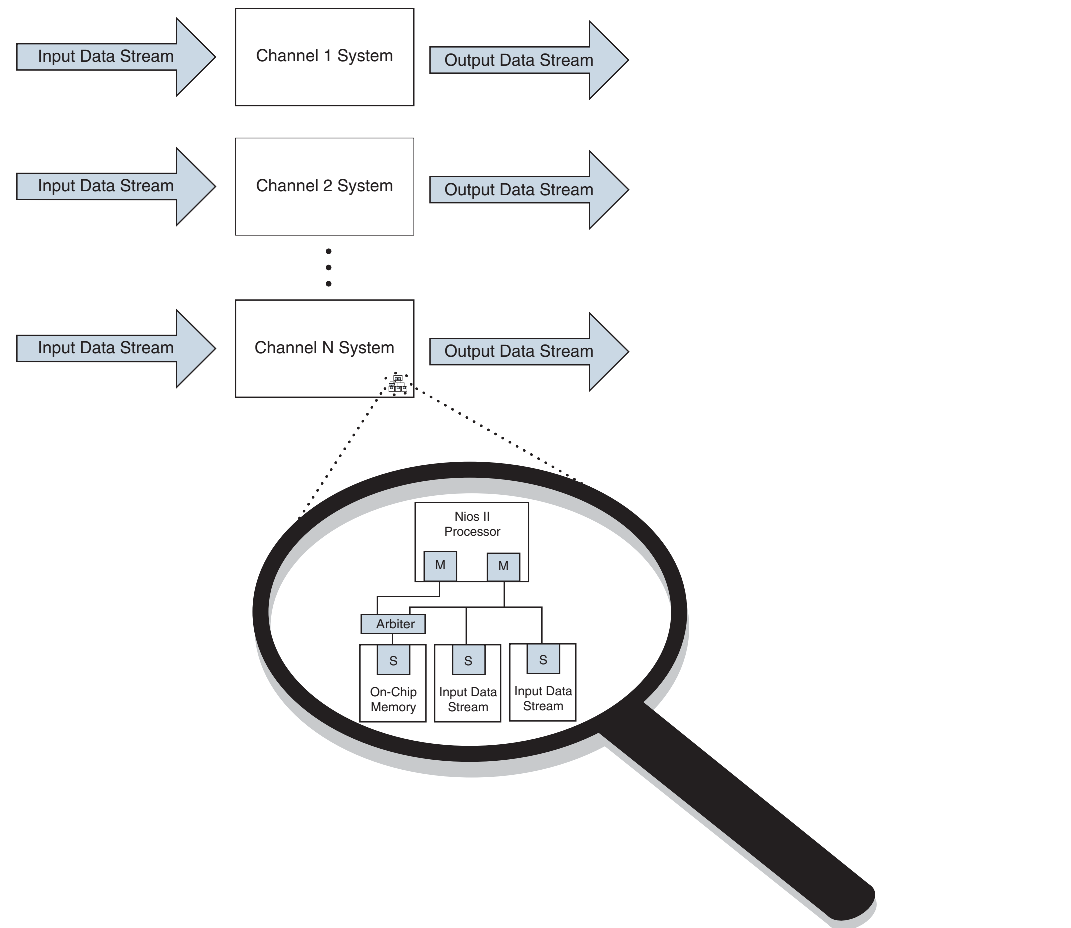
For more information, refer to *Creating Multiprocessor Nios II Systems Tutorial* and *Multiprocessor Coordination Peripherals*.

Figure 6–8. Message Passing



Alternatively, if the subsystems are identical, you can design a single SOPC Builder system and instantiate it multiple times in your FPGA design. This approach has the advantage of being easier to maintain than a heterogeneous system. In addition, such systems are easier to scale because once you know the logic utilization and efficiency of a single instance, you can estimate how much logic is necessary for multiple subsystems. Systems that process multiple data channels are frequently designed by instantiating the same sub-system for each channel.

Figure 6–9. Multi-Channel System



Increasing Transfer Throughput

Increasing the transfer efficiency of the master and slave ports in your system increases the throughput of your design. Designs with strict cost or power requirements benefit from increasing the transfer efficiency because less expensive, lower frequency devices can be used. At the other end of the spectrum, designs requiring high performance also benefit from increased transfer efficiency because it improves the performance of frequency–limited hardware.

Using Pipelined Transfers

Pipelined transfers increase the read efficiency by allowing a master to post multiple reads before data from the earlier reads returns. Masters that support pipelined transfers post transfers continuously, relying on the `readdatavalid` signal to indicate valid data. Slaves support pipelined transfers by including the `readdatavalid` signal or operating with a fixed read latency.

Maximum Pending Reads

SOPC Builder updates the maximum pending reads property when it generates the system interconnect fabric. If you create a custom component with a slave port supporting reads, you must specify this value in the Component Editor. This value represents the maximum number of read transfers your pipelined slave component can handle. If the number of reads presented to the slave port exceeds the maximum pending reads parameter your component must assert `waitrequest`.

Selecting the Maximum Pending Reads Value

Optimizing the value of the maximum pending reads parameter requires a good understanding of the latencies of your custom components. This parameter should be based on the component's longest delay. For example, if your pipelined component has two modes, one requiring two clock cycles and the other five, set the maximum pending reads value to five. Doing so allows your component to pipeline five transfers, eliminating dead cycles after the initial five-cycle latency.

Another way to determine the correct value for the maximum pending reads parameter is to monitor the number of reads that are actually pending during system simulation or while running the actual hardware. To use this method, set the maximum pending reads to a very high value and use a master that issues read requests on every clock. You can use a DMA for this task as long as the data is written to a location that does not assert `waitrequest` frequently. If you run this experiment with the actual hardware, you can use a logic analyzer or built-in monitoring hardware to observe your component.

Overestimating Versus Underestimating the Maximum Pending Reads Value

Choosing the correct value for the maximum pending reads value of your custom pipelined read component is very important. If you underestimate the maximum pending reads value you either loose data or cause a master port to stall indefinitely. The system interconnect fabric has no timeout mechanism to handle long delays.

The maximum pending reads value dictates the depth of the `readdatavalid` FIFO inserted into the system interconnect fabric for each master connected to the slave. Because this FIFO is only one bit wide it does not consume significant hardware resources. Overestimating the maximum pending reads value for your custom component results in a slight increase in hardware utilization. For these reasons, if you are not sure of the optimal value, it is better to overestimate this value.

Pipelined Read Masters

A pipelined read master can post multiple reads before data returns. Pipelined read masters hide the latency of read operations by posting reads as frequently as every clock cycle. A pipelined read master can prefetch data when no data dependencies are present. Examples of common pipelined read masters include the following:

- DMA engines
- Nios II processor (with a cache line size greater than four bytes)
- C2H read masters

Requirements

The logic for the control and datapaths of pipelined read masters must be carefully designed. The control logic must extend a read cycle whenever the `waitrequest` signal is asserted. This logic must also control the master `address`, `byteenable`, and `read` signals. To achieve maximum throughput, pipelined read masters should post reads continuously as long as `waitrequest` is deasserted. While `read` is asserted the address presented to the system interconnect fabric is stored.

The datapath logic includes the `readdata` and `readdatavalid` signals. If your master can return data on every clock cycle, you can register the data using `readdatavalid` as the enable bit. If your master cannot handle a continuous stream of read data, it must buffer the data in a FIFO. The control logic must stop issuing reads when the FIFO reaches a predetermined fill level.

Refer to the *Avalon Interface Specifications* to learn more about the signals that implement a pipelined read master.

Throughput Improvement

The throughput gain that you achieve by using a pipelined read master is typically directly proportional to the latency of the slave port. For example, if the read latency is two cycles, you can double your throughput using a pipelined read master, assuming the slave port also supports pipeline transfers. If either the master or slave does not support pipelined read transfers then the system interconnect fabric asserts `waitrequest` until the transfer completes.

When both the master and slave ports support pipelined read transfers, data flows in a continues stream after the initial latency. Figure 6–10 illustrates the case where reads are not pipelined. The system pays a penalty of 3 cycles latency for each read, making the overall throughput 25 percent. Figure 6–11 illustrates the case where reads are pipelined. After the initial penalty of 3 cycles latency, the data flows continuously.

Figure 6–10. Low Efficiency Read Transfer

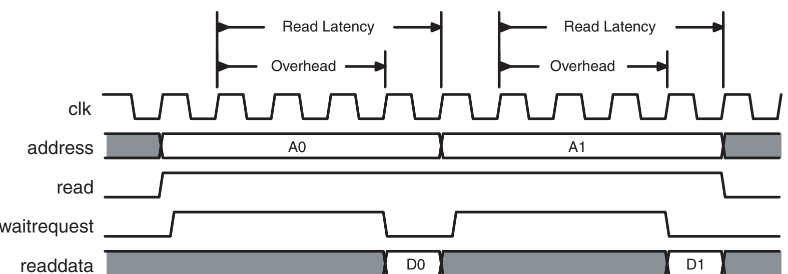
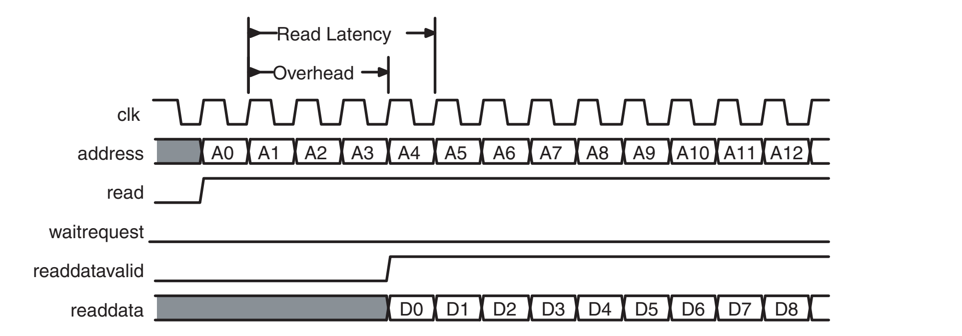


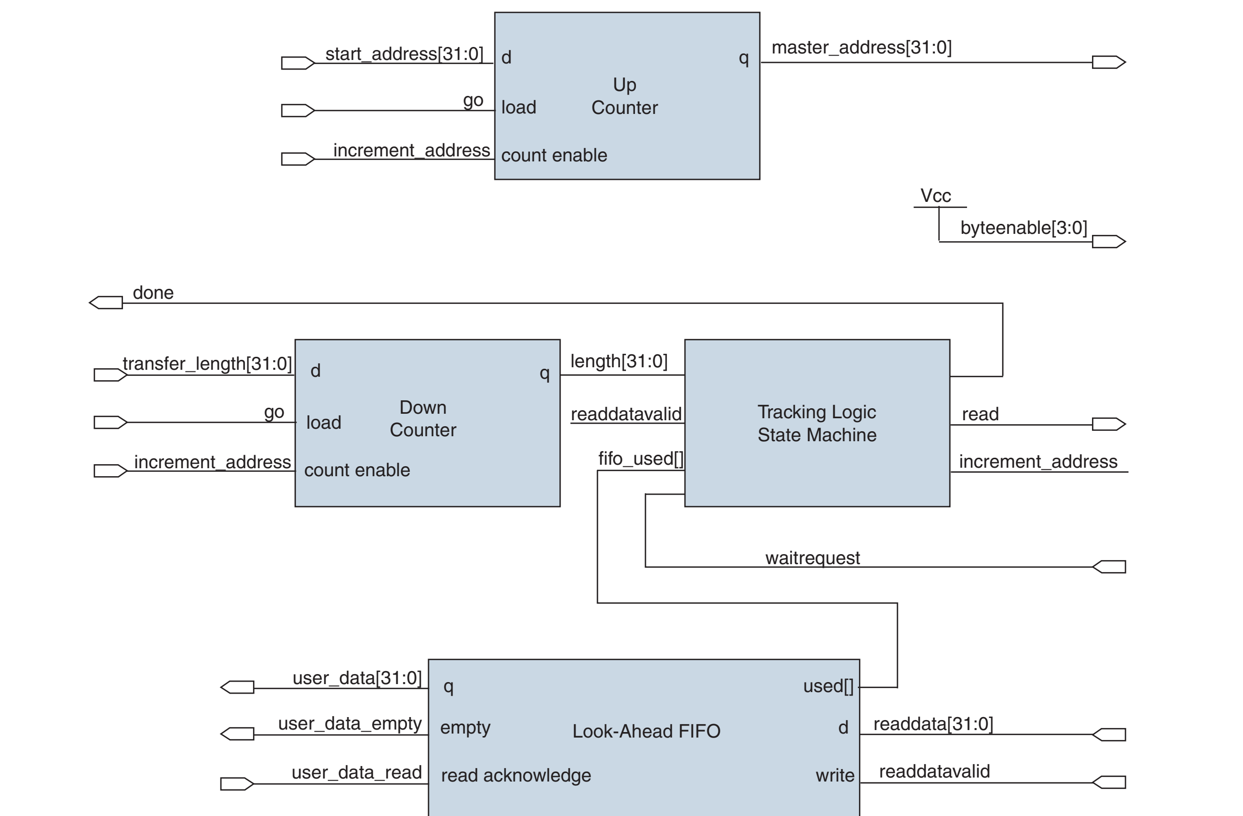
Figure 6–11. High Efficiency Read Transfer



Pipelined Read Master Example

Figure 6–12 illustrates a pipelined read master that stores data in a FIFO that can be used to implement a custom DMA, hardware accelerator, or off-chip communication interface. To simplify the design, the control and data logic are separate. The master performs word accesses that are word-aligned and reads from sequential memory addresses. The transfer length is a multiple of the word size.

Figure 6–12. Latency Aware Master



When the go bit is asserted, the master registers the start_address and transfer_length signals. The master begins issuing reads on the next clock and does not stop until the length register reaches zero. In this example, the word size is 4 bytes so that the address always increments by 4 and the length decrements by 4. The read signal remains asserted unless the FIFO fills to a predetermined level. The address register increments and the length register decrements if the length has not reached 0 and a read is posted.

The master posts a read transfer every time the read signal is asserted and the waitrequest is deasserted. The master issues reads until the entire buffer has been read or waitrequest is asserted. An optional tracking block times the done bit. When the length register reaches 0, some reads will be outstanding. The tracking logic guarantees that done is not asserted until the last read completes. The tracking logic monitors the number of reads posted to the system interconnect fabric so that is does not exceed the space remaining in the readdata FIFO. This logic includes a counter that counts if the following conditions are met:

- If a read is posted and readdatavalid is deasserted, the counter increments.
- If a read is not posted and readdatavalid is asserted, the counter decrements.

When the length register and the tracking logic counter reach 0, all the reads have completed and the done bit is asserted. The done bit is important if a second master overwrites the memory locations that the pipelined read master accesses. This bit guarantees that all the reads have completed before the original data is overwritten.

To learn more about creating Avalon-MM masters refer to the following design examples and documentation:

- *Nios II Embedded Processor Design Examples*
- *Developing Components for SOPC Builder* in volume 4 of the *Quartus II Handbook*.

Arbitration Shares and Bursts

Arbitration shares provide control over the arbitration process. By default, the arbitration algorithm provides equal fairness, with all masters receiving one share. You can tune the arbitration process to your system requirements by assigning a larger number of shares to the masters that need greater throughput. The larger the arbitration share, the more transfers are allocated to the master to access a slave.

If a master cannot post a transfer and other masters are waiting to gain access to a particular slave, the arbiter grants another master access in a round robin fashion. This mechanism prevents a master from using arbitration cycles if it cannot post back-to-back transfers.

Bursts allow a master to maintain access to a slave for more than a single word transfer. If a bursting master posts a write transfer with a burst length of 8, it is guaranteed arbitration for 8 write cycles.

Differences between Arbitration Shares and Bursts

The following three key characteristics distinguish between arbitration shares and bursts:

- Arbitration lock
- Sequential addressing

- Burst adapters

Arbitration Lock

When a master posts a burst transfer, the arbitration is locked for that master; consequently, the bursting master should be capable of sustaining transfers for the duration of the locked period. If, after the fourth write, the master deasserts the `write` signal for fifty cycles, all other masters continue to wait for access during this stalled period.

To avoid wasting bandwidth, your master designs should wait until a full burst transfer is ready before requesting access to a slave device. Alternatively, you can avoid wasted bandwidth by posting burst counts equal to the amount of data that is ready. For example, if you have created a custom bursting write master with a maximum burst count of 8, but only 3 words of data are ready, you can simply present a burst count of 3. This strategy does not result in optimal use of the system bandwidth; however, it prevents starvation for other masters in the system.

Sequential Addressing

By definition, a burst transfer includes a base address and a burst count. The burst count represents the number of words of data to be transferred starting from the base address and incrementing sequentially. Burst transfers are common for processors, DMAs, and buffer processing accelerators; however, there are occasions when a master must access non-sequential addresses. Consequently, a bursting master must set the burst count to the number of sequential addresses and then reset the burst count for the next location.

The arbitration share algorithm has no restrictions on addresses; therefore, your custom master can update the address it presents to the system interconnect fabric for every read or write transaction.

Burst Adapters

SOPC Builder allows you to create systems that mix bursting and non-bursting master and slave ports. It also allows you to connect bursting master and slave ports that support different maximum burst lengths. In order to support all these cases, SOPC Builder generates burst adapters when appropriate.

SOPC Builder inserts a burst adapter whenever a master port burst length exceeds the burst length of the slave port. SOPC Builder assigns non-bursting masters and slave ports a burst length of one. The burst adapter divides long bursts into shorter bursts. As a result, the burst adapter adds logic to the address and burstcount paths between the master and slave ports.

Choosing Interface Types

To avoid inefficient transfers, custom master or slave ports must use the appropriate interfaces. The are three possible interface types: simple, pipelined and burst. Each is described below:

Simple

Simple interfaces do not support pipelining or bursting for reads or writes; consequently, their performance is limited. Simple interfaces are appropriate for transfers between masters and infrequently used slave ports. In SOPC Builder, the PIO, UART, and Timer include slave ports that operate at peak efficiency using simple transfers.

When designing a custom component, Altera recommends that you start with a simple interface. If performance becomes an issue, you can modify the component to support either pipelined reads or bursting.

Pipelined

In many systems, read throughput becomes inadequate if simple reads are used. If your system requires high read throughput and is not overly sensitive to read latency, then your component can implement pipelined transfers. If you define a component with a fixed read latency, SOPC Builder automatically provides the logic necessary to support pipelined reads. If your component has a variable latency response time, use the readdatavalid signal to indicate valid data. SOPC Builder implements a `readdatavalid` FIFO to handle the maximum number of pending read requests.

To use components that support pipelined read transfers efficiently, your system must contain pipelined masters. Refer to the "Pipelined Read Master Example" on page 6–15 for an example of a pipelined read master.

Burst

Burst transfers are commonly used for latent memories and off-chip communication interfaces. To use a burst-capable slave port efficiently, you must connect it to a bursting master. Components that require bursting to operate efficiently typically have an overhead penalty associated with short bursts or non-bursting transfers.

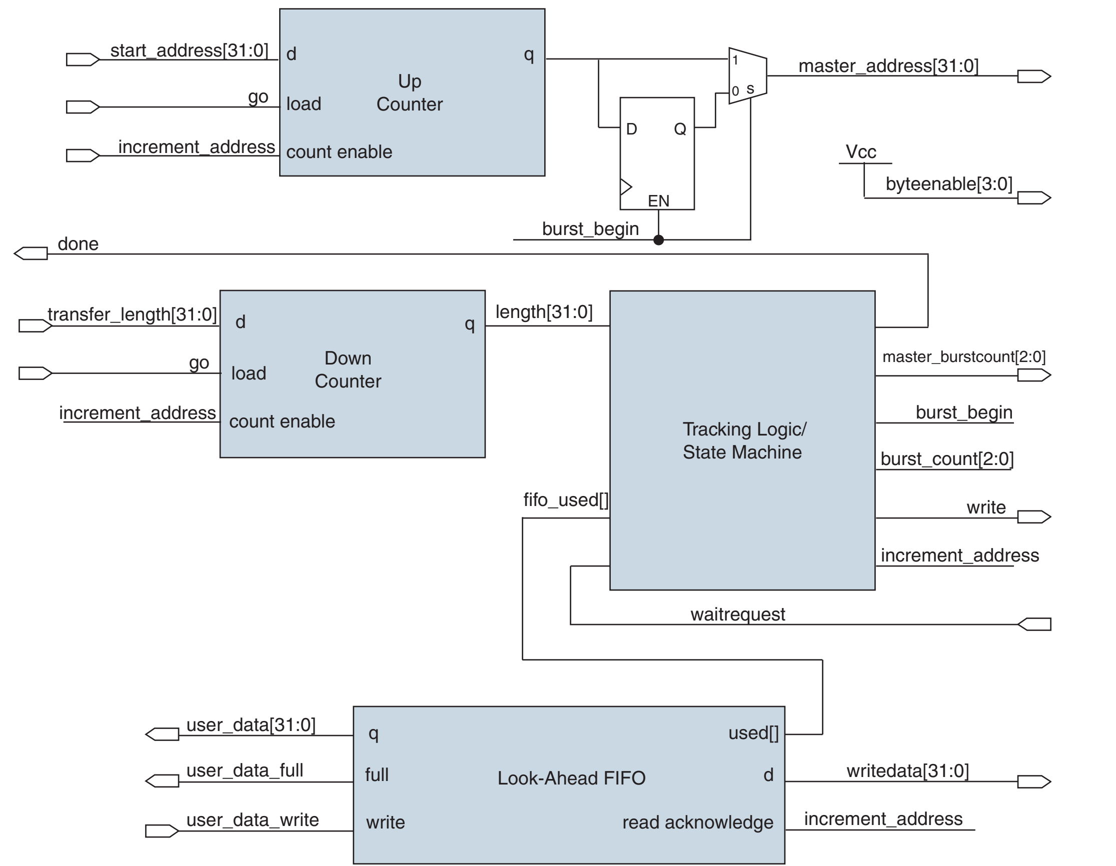
Altera recommends that you design a burst-capable slave port if you know that your component requires sequential transfers to operate efficiently. Because DDR SDRAM memories incur a penalty when switching banks or rows, performance improves when they are accessed sequentially using bursts.

Any shared address and data bus architecture also benefits from bursting. Whenever an address is transferred over a shared address and data bus, the throughput of the data transfer is reduced. Because the address phase adds overhead, using large bursts increases the throughput of the bus.

Burst Master Example

Figure 6–13 illustrates the architecture of a bursting write master that receives data from a FIFO and writes the contents to memory. You can use this master as a starting point for your own bursting components, such as custom DMAs, hardware accelerators, or off-chip communication interfaces. In Figure 6–13, the master performs word accesses and writes to sequential memory locations.

Figure 6–13. Bursting Write Master



When `go` is asserted, the `address` and `length` are registered. On the following clock cycle, the control logic asserts `burst_begin`. The `burst_begin` signal synchronizes the internal control signals in addition to the `master_address` and `master_burstcount` presented to the system interconnect fabric. The timing of these two signals is important because during bursting write transfers `address`, `byteenable`, and `burstcount` must be held constant for the entire burst.

To avoid inefficient writes, the master only posts a burst when enough data has been buffered in the FIFO. To maximize the burst efficiency, the the master should only stall when a slave asserts `waitrequest`. In this example the FIFO's `used` signal tracks the number of words of data that are stored in the FIFO and determines when enough data has been buffered.

The `address` register increments after every word transfer, and the `length` register decrements after every word transfer. The address remains constant throughout the burst. Because a transfer is not guaranteed to complete on burst boundaries, additional logic is necessary to recognize the completion of short bursts.

The *Accelerated FIR with Built-in Direct Memory Access Example*, includes a pipelined read master and bursting write master similar to those used in Figure 6–13.

Increasing System Frequency

In SOPC Builder, you can introduce bridges to reduce the amount of logic that SOPC Builder generates and increase the clock frequency.

In SOPC Builder, you can use bridges to control the system interconnect topology. Bridges allow you to subdivide the system interconnect fabric, giving you more control over pipelining and clock crossing functionality.

This section assumes that you have read *Avalon Memory-Mapped Bridges* chapter in volume 4 of the *Quartus II Handbook*. To see an example of a design containing bridges, refer to the *Nios II High-Performance Example With Bridges*.

Use Pipeline Bridges

The pipeline bridge contains Avalon-MM master and slave ports. Transfers to the bridge slave port are propagated to the master port which connects to components downstream from the bridge. You have the option to add the following pipelining features between the bridge ports:

- Master-to-Slave Pipelining
- Slave-to-Master Pipelining
- waitrequest Pipelining

The pipeline bridge options can increase your logic utilization and read latency. As a result, you should carefully consider the effects of the following options described in this section.

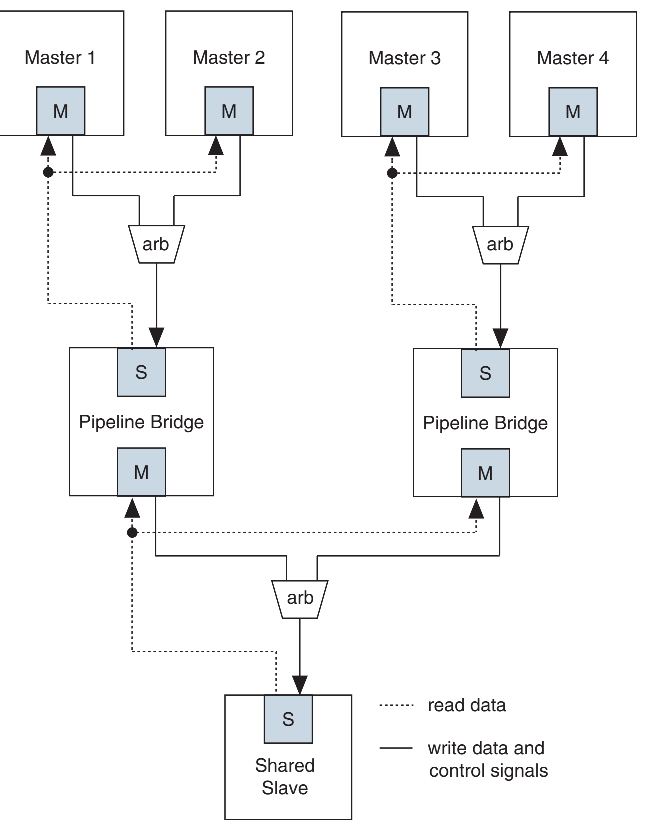
Master-to-Slave Pipelining

Master-to-slave pipelining is advantageous when many masters share a slave device. The arbitration logic for the slave port must multiplex the `address`, `writedata`, and `burstcount` signals. The multiplexer width increases as the number of masters connecting to a single slave port increases, causing timing problems if the slave component does not register the input signals.

This option is helpful if the `waitrequest` signal becomes part of a critical timing path. Because `waitrequest` is dependent on arbitration logic, which is driven by the master address, enabling master-to-slave pipelining helps pipeline this path. If the `waitrequest` signal remains a part of a critical timing path, you should explore using the `waitrequest` pipelining feature of the pipelined bridge.

If a single pipeline bridge provides insufficient improvement, you can instantiate this bridge multiple times, in a binary tree structure, to increase the pipelining and further reduce the width of the multiplexer at the slave port as Figure 6–14 illustrates.

Figure 6–14. Tree of Bridges



Slave-to-Master Pipelining

Slave-to-master pipelining is advantageous for masters that connect to many slaves that support read transfers. The system interconnect fabric inserts a multiplexer for every read datapath back to the master. As the number of slaves supporting read transfers connecting to the master increases, so does the width of the read data multiplexer. As with master-to-slave pipelining, if the performance increase is insufficient, you can use multiple bridges to improve f_{MAX} using a binary tree structure.

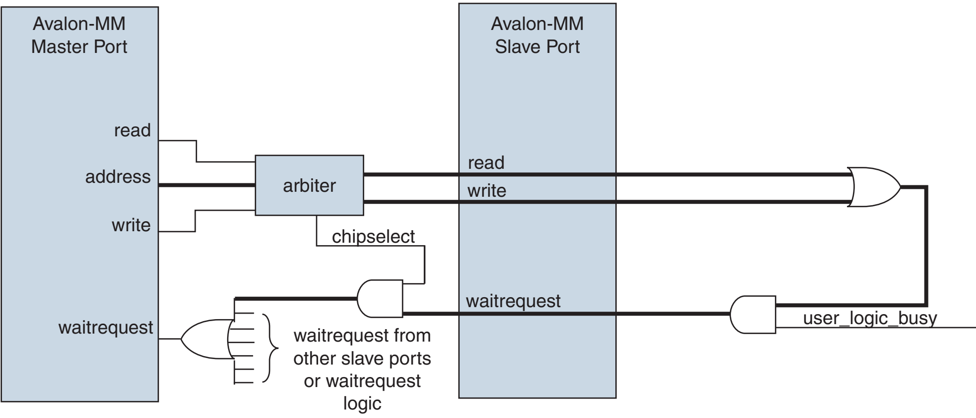
waitrequest Pipelining

`waitrequest` pipelining can be advantageous in cases where a single master connects to many slaves. Because slave components and system interconnect fabric drive `waitrequest`, the logic depth grows as the number of slaves connected to the master increases. `waitrequest` pipelining is also useful when multiple masters connect to a single slave, because it pipelines the arbitration logic.

In many cases `waitrequest` is a combinational signal because it must be asserted during the same cycle that a `read` or `write` transaction is posted. Because `waitrequest` is typically dependent on the master `read` or `write` signals, it creates a timing path from the master to the slave and back to the master. Figure 6–15 illustrates this round-trip path with the thick wire.

To prevent this round-trip path from impacting the f_{MAX} of your system, you can use master-to-slave pipelining to reduce the path length. If the `waitrequest` signal remains part of the critical timing path, you can consider using the `waitrequest` pipelining feature. Another possibility is to register the `waitrequest` signal and keep it asserted, even when the slave is not selected. When the slave is selected, it has a full cycle to determine whether it can respond immediately.

Figure 6–15. Typical Slow Waitrequest Path



Use a Clock Crossing Bridge

The clock crossing bridge contains an Avalon-MM master port and an Avalon-MM slave port. Transfers to the slave port are propagated to the master port. The clock crossing bridge contains a pair of clock crossing FIFOs which isolate the master and slave interfaces in separate, asynchronous clock domains.

Because FIFOs are used for the clock domain crossing, you gain the added benefit of data buffering when using the clock crossing bridge. The buffering allows pipelined read masters to post multiple reads to the bridge even if the slaves downstream from the bridge do not support pipelined transfers.

Increasing Component Frequencies

One of the most common uses of the clock crossing bridge is to place high and low frequency components in separate clock domains. If you limit the fast clock domain to the portion of your design that requires a high performance, you can achieve a higher f_{MAX} for this portion of the design.

Reducing Low-Priority Component Frequencies

The majority of components included in embedded designs do not benefit from operating at higher frequencies. Examples components that do not require high frequencies are timers, UARTs, and JTAG controllers.

When you compile a design using the Quartus II design software, the fitter places your logic into regions of the FPGA. The higher the clock frequency of your system, the longer a compilation takes. The compilation takes more time because the fitter needs more time to place registers to achieve the required f_{MAX}. To reduce the amount of effort that the fitter uses on low priority and low performance components, you can place these behind a clock crossing bridge operating at a lower frequency, allowing the fitter to increase the effort placed on the higher priority and higher frequency datapaths.

Consequences of Using Bridges

Before using the pipeline or clock crossing bridges in your design, you should carefully consider their effects. The bridges can have any combination of the following effects on your design:

- Increased Latency
- Limited Concurrency
- Address Space Translation

Depending on your system, these effects could be positive or negative. You can use benchmarks to test your system before and after inserting bridges to determine their effects. The following sections discuss the effects of adding bridges.

Increased Latency

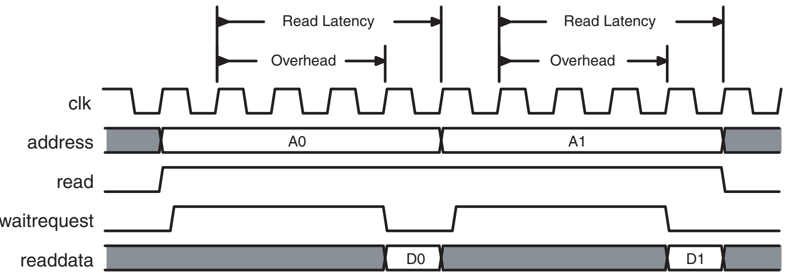
Adding either type of bridge to your design has an effect on the read latency between the master and the slave. Depending on the system requirements and the type of master and slave this latency increase may or may not be acceptable in your design.

Acceptable Latency Increase

For the pipeline bridge, a cycle of latency is added for each pipeline option that is enabled. The buffering in the clock crossing bridge also adds latency. If you use a pipelined or burst master that posts many read transfers, the increase in latency does not impact performance to a significant degree because it is very small compared to the length data transfer.

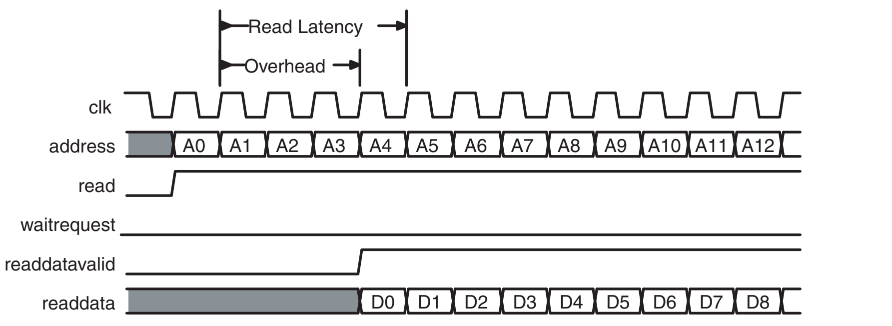
For example, if you use a pipelined read master such as a DMA controller to read data from a component with a fixed read latency of 4 clock cycles but only perform a single word transfer, the overhead is 3 clock cycles out of the total 4. The read throughput is only 25%.

Figure 6–16. Low Efficiency Read Transfer



On the other hand, if 100 words of data are transferred without interruptions, the overhead is 3 cycles out of the total of 103 clock cycles, corresponding to a read efficiency of approximately 97%. Adding a pipeline bridge to this read path adds two extra clock cycles of latency. The transfer requires 105 cycles to complete, corresponding to an efficiency of approximately 94%. Although the efficiency decreased by 3%, adding the bridge increases the f_{MAX} by 5%. The overall throughput improves. As the number of words transferred increases the efficiency will increase to approach 100%, whether or not a pipeline bridge is present.

Figure 6–17. High Efficiency Read Transfer

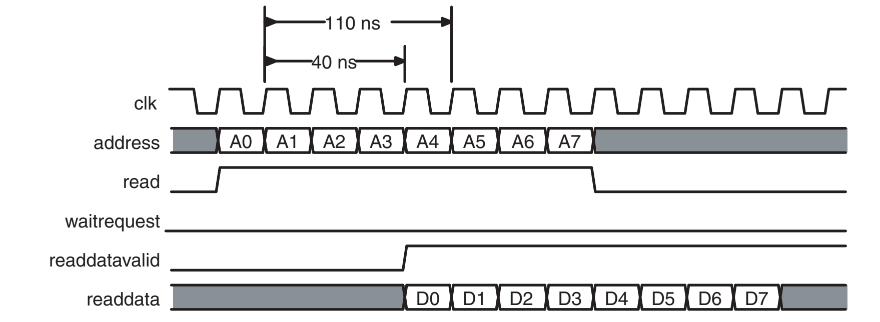


Unacceptable Latency Increase

Processors are sensitive to high latency read times. They typically fetch data for use in calculations that cannot proceed until it arrives. Before adding a bridge to the datapath of a processor instruction or data master, determine whether the clock frequency increase justifies the added latency.

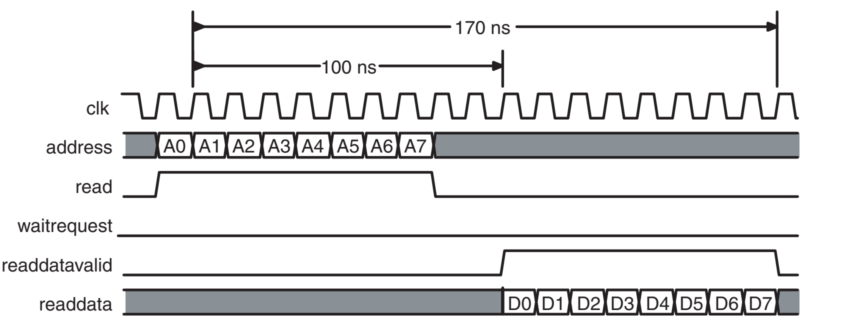
The following example design illustrates this point. The original design contains a Nios II processor and memory operating at 100 MHz. The Nios II processor instruction master has a cache memory with a read latency of 4 cycles. Eight sequential words of data return for each read. At 100 MHz, the first read takes 40 ns to complete. Each successive word takes 10 ns so that 8 reads complete in 110 ns.

Figure 6–18. Eight Reads with Four Cycles Latency



Adding a clock crossing bridge allows the memory to operate 125 MHz. However, this increase in frequency is negated by the increase in latency for the following reasons. Assume that the clock crossing bridge adds 6 clock cycles of latency at 100 MHz. The memory still operates with a read latency of 4 clock cycles; consequently, the first read from memory takes 100 ns and each successive word takes 10 ns because reads arrive at the processor's frequency, 100 MHz. In total, all 8 reads complete after 170 ns. Although the memory operates at a higher clock frequency, the frequency at which the master operates limits the throughput.

Figure 6–19. Eight Reads with Ten Cycles latency

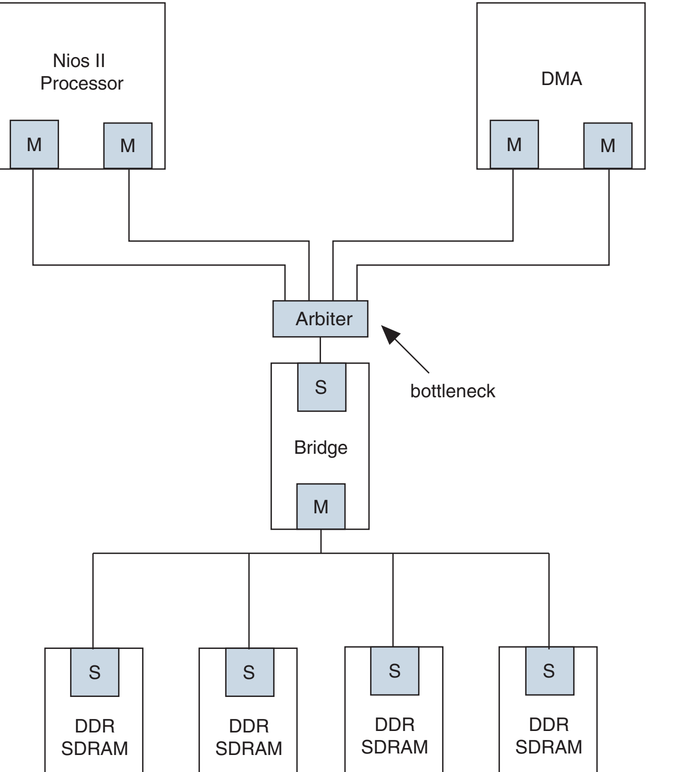


Limited Concurrency

Placing an Avalon bridge between multiple Avalon-MM master and slave ports limits the number of concurrent transfers your system can initiate. This limitation is no different than connecting multiple master ports to a single slave port. The bridge's slave port is shared by all the masters and arbitration logic is created as a result. If the components placed behind a bridge are infrequently accessed, this concurrency limitation may be acceptable.

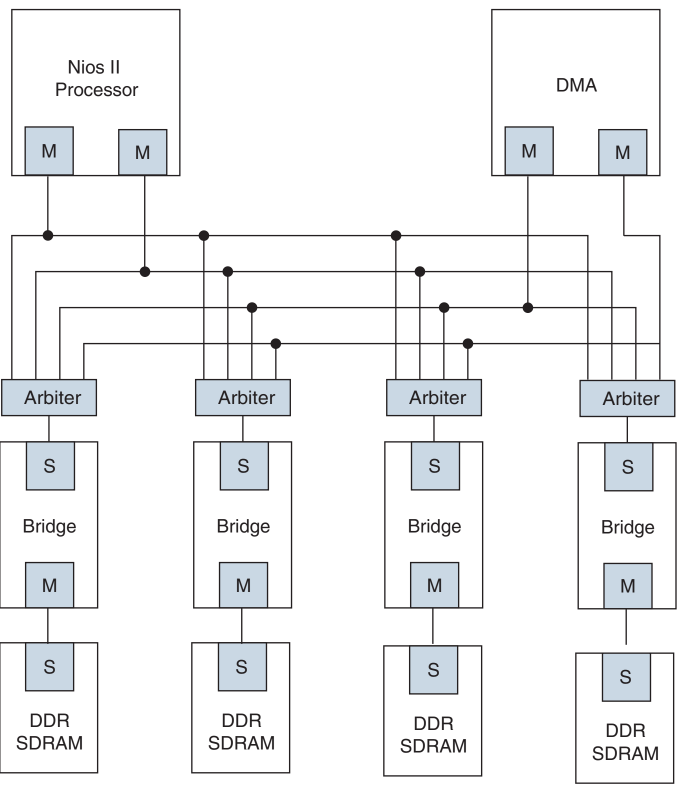
Bridges can have a severe negative impact on system performance if used inappropriately. For instance, if multiple memories are used by several masters, you should not place them all behind a bridge. The bridge limits the memory performance by preventing concurrent memory accesses. Placing multiple memory components behind a bridge, causes the separate slave interfaces to appear as one monolithic memory to the masters accessing the bridge; they must all access the same slave port. Figure 6–20 illustrates this configuration.

Figure 6–20. Poor Memory Pipelining



If the f_{MAX} of your memory interfaces is low, you can place each memory behind its own bridge, which increases the f_{MAX} of the system without sacrificing concurrency as Figure 6–21 illustrates.

Figure 6–21. Efficient Memory Pipelining



Address Space Translation

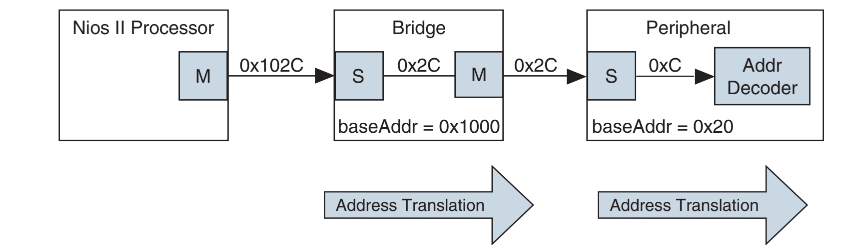
The slave port of a pipeline or clock crossing bridge has a base address and address span. You can set the base address or allow SOPC Builder to set it automatically. The slave port's address is the base offset address of all the components connected to the bridge. The address of components connected to the bridge is the sum of the base offset and that component's address. The address span of the bridge is automatically calculated by SOPC Builder based on the address range of all the components connected to it.

Address Shifting

The master port of the bridge only drives the address bits that represent the offset from the base address of the bridge slave port. Any time an Avalon-MM master accesses a slave through a bridge, both addresses must be added together, otherwise the transfer fails. Clicking the **Address Map** button in SOPC Builder displays the addresses of the slaves connected to each master taking into account any address translations caused by bridges in the system.

Figure 6–22 illustrates how this address translation takes place. In this example, the Nios II processor connects to a bridge located at base address 0x1000. A slave connects to the bridge master port at an offset of 0x20 and the processor performs a write transfer to the fourth 32-bit word within the slave. Nios II drives the address 0x102C to system interconnect fabric which lies within the address range of the bridge. The bridge master port drives 0x2C which lies within the address range of the slave and the transfer completes

Figure 6–22. Avalon Bridge Address Translation

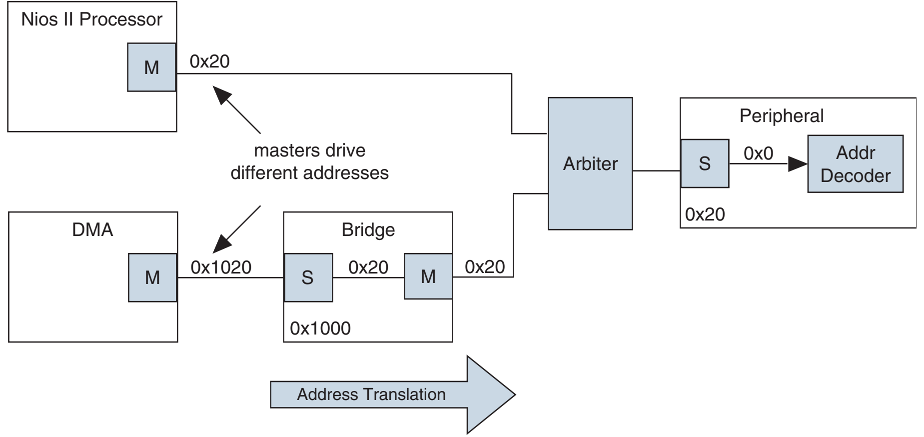


Address Coherency

To avoid unnecessary complications in software, all masters should access slaves at the same location. In many systems a processor passes buffer locations to other mastering components such as a DMA controller. If the processor and DMA controller do not access the slave at the same location, software must compensate for the differences.

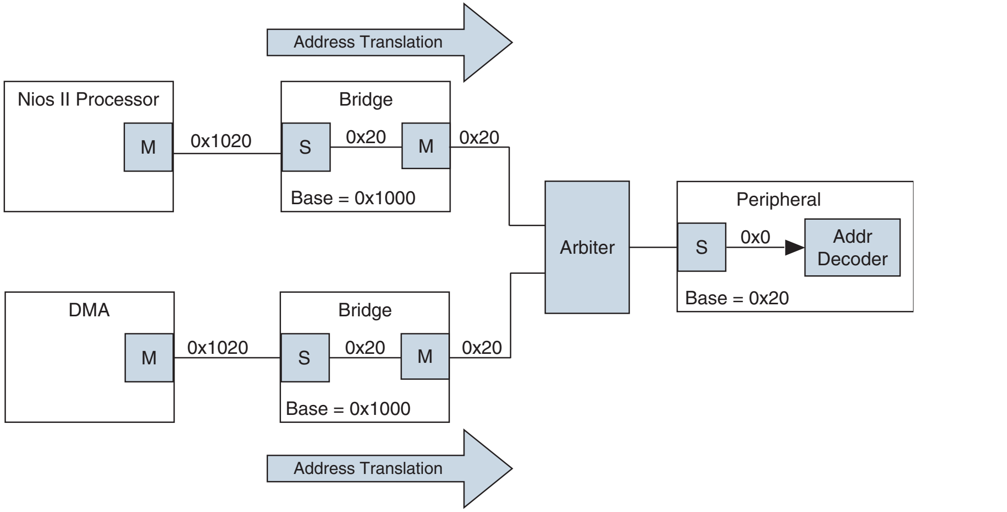
In the following example, a Nios II processor and DMA controller access a slave port located at address 0x20. The processor connects directly to the slave port. The DMA controller connects to a pipeline bridge located at address 0x1000 which then connects to the slave port. Because the DMA controller accesses the pipeline bridge first, it must drive 0x1020 to access the first location of the slave port. Because the processor accesses the slave from a different location, the software developer must maintain two base addresses for the slave device.

Figure 6–23. Slave at Different Addresses, Complicating the Software



To avoid this issue, you can add an additional bridge to the design and set its base address to 0x1000. You can disable all the pipelining options in this second bridge so that it has a minimal impact on the system timing and resource utilization. Because this second bridge has the same base address as the bridge the DMA controller connects to, both the processor and DMA controller access the slave port using the same address range.

Figure 6–24. Address Translation Corrected Using Bridge



Minimize System Interconnect Logic

In SOPC Builder, you have control over the address space of your system, as well as the connections between master sand slaves. This control allows you to make minor changes to your system in order to increase the overall system performance. The following sections explain design changes you can make to improve the f_{MAX} of your system.

- Use Unique Address Bits
- Create Dedicated Master and Slave Connections
- Remove Unnecessary Connections

Use Unique Address Bits

For every slave in your system, SOPC Builder inserts comparison logic to drive the arbiter to select a slave. This comparison logic determines if the master is performing an access to the slave port by determining if the address presented is in the slave port's address range. This result is ANDed with the master read and write signals to determine if the transfer is destined for the slave port.

The comparison logic can become part of a failing timing path because the result is used to drive the slave port. To reduce this path length, you can move the slave port base address to use unique MSBs for the comparison. Frequently, you can reduce the comparison logic to a single logic element if you avoid using a base address that shares MSBs with other slave ports.

Consider a design with 4 slave ports each having an address range of 0x10 bytes connected to a single master. If you use the **Auto-Assign Base Addresses** option in SOPC Builder, the base addresses for the 4 slaves is set to 0x0, 0x10, 0x20, and 0x30, which corresponds to the following binary numbers: 6b'000000, 6b'010000, 6'b100000, and 6b'110000. The two MSBs must be decoded to determine if a transfer is destined for any of the slave ports.

If the addresses are located at 0x10, 0x20, 0x40, and 0x80, no comparison logic is necessary. These binary locations are: 6'b00010000, 6b'00100000, 6b'01000000, and 6b'10000000. This technique is referred to as *one-hot* encoding because a single asserted address bit replaces comparison logic to determine if a slave transfer is taking place. In this example, the performance gained by moving the addresses would be minimal; however, when you connect many slave ports of different address spans to a master this technique can result in a significant improvement.

Create Dedicated Master and Slave Connections

In some circumstances it is possible to modify a system so that a master port connects to a single slave port. This configuration eliminates address decoding, arbitration, and return data multiplexing, greatly simplifying the system interconnect fabric. Dedicated master-to-save connections attain the same clock frequencies as Avalon-ST connections with the added benefits offered by Avalon-MM.

Typically these one-to-one connections include an Avalon-MM bridge or hardware accelerator. For example, if you insert a pipeline bridge between a slave and all other master ports, the logic between the bridge master and slave port is reduced to wires. Figure 6–21 illustrates this technique. If a hardware accelerator only connects to a dedicated memory, no system interconnect logic is generated between the master and slave pair.

Remove Unnecessary Connections

The number of connections between master and slave ports has a great influence on the f_{MAX} of your system. Every master port that you connect to a slave port increases the multiplexer select width. As a multiplexer width increases, so does the logic depth and width that implements the multiplexer in the FPGA. To improve your system performance, only connect masters and slaves when necessary.

In the case of a master port connecting to many slave ports, the multiplexer for the `readdata` signal grows as well. Use bridges to help control this depth of multiplexers, as Figure 6–14 illustrates.

Reducing Logic Utilization

The system interconnect fabric supports the Avalon-MM and Avalon-ST interfaces. Although the system interconnect fabric for Avalon-ST interfaces is lightweight, the same is not always true for the Avalon-MM. This section describes design changes you can make to reduce the logic footprint of the system interconnect fabric.

Minimize Arbitration Logic by Consolidating Components

As the number of components in your design increases, so does the amount logic required to implement the system interconnect fabric. The number of arbitration blocks increases for every slave port that is shared by multiple master ports. The width of the `readdata` multiplexer increases as the number of slave ports supporting `read` transfers increases on a per master port basis. For these reasons you should consider implementing multiple blocks of logic as a single component to reduce the system interconnect fabric logic utilization.

Logic Consolidation Tradeoffs

Consider the following two tradeoffs before making any modifications to your system or components. First, consider the impact on concurrency that consolidating components has. When your system has four master components and four slave components, it can initiate four concurrent accesses. If you consolidate all four slave components into a single component, all four masters must compete for access. Consequently, you should only combine low priority components such as low speed parallel I/O devices where the combination will not impact the performance.

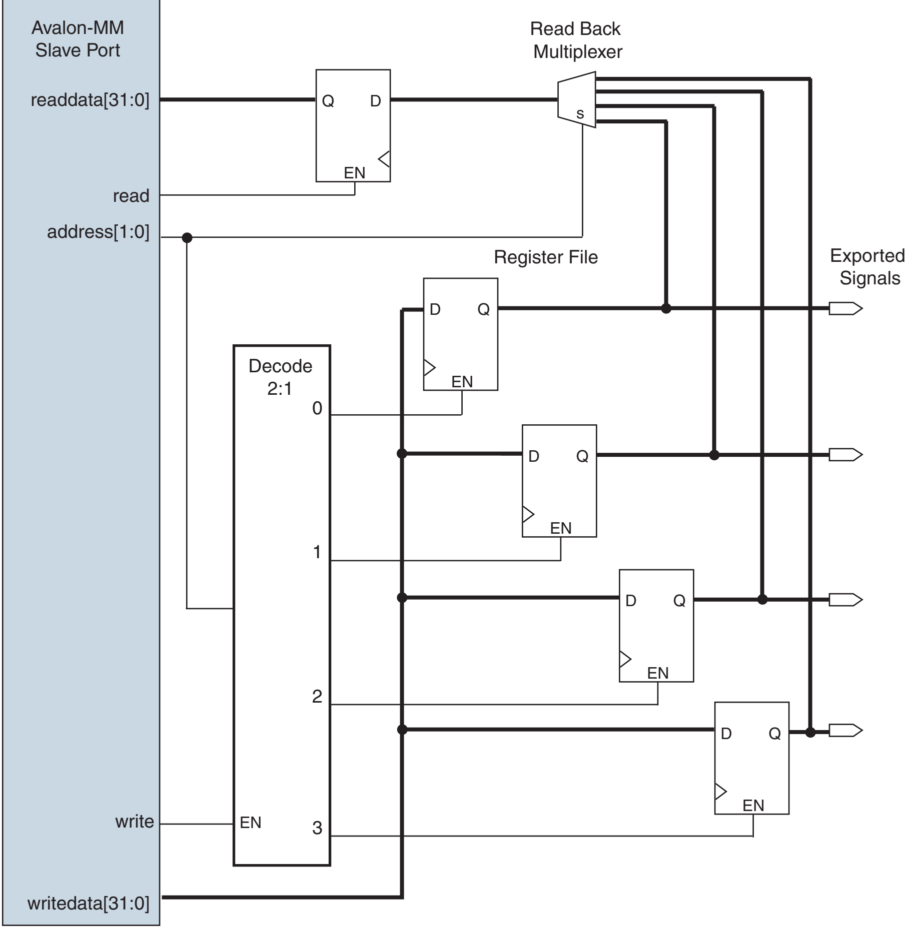
Second, determine whether consolidation introduces new decode and multiplexing logic for the slave port that the system interconnect fabric previously included. If a component contains multiple read and write address locations it already contains the necessary decode and multiplexing logic. When you consolidate components, you typically reuse the decoder and multiplexer blocks already present in one of the original components; however, it is possible that combining components will simply move the decode and multiplexer logic, rather than eliminating duplication.

Combined Component Example

Figure 6–25 illustrates set of four output registers that support software read back. The registers can be implemented using four PIO components in SOPC Builder; however, this example provides a more efficient implementation of the same logic. You can use this example as a template to implement any component that contains multiple registers mapped to memory locations.

Components that implement reads and writes require three main building blocks: an address decoder, a register file, and a read multiplexer. In this example, the read data is a copy of the register file outputs. The read back functionality may seem redundant; however, it is useful for verification.

Figure 6–25. Four PIOs



The decoder enables the appropriate 32-bit PIO register for writes. For reads, the address bits drive the multiplexer selection bits. The `read` signal registers the data from the multiplexer so that the component can achieve a high clock frequency. In the SOPC Builder component editor, this component would be described as having 0 write wait states and 1 read wait state. Alternatively, you could set both the read and write wait states to 0 and specify a read latency of 1 because this component also supports pipelined reads.

Use Bridges to Minimize System Interconnect Fabric Logic

Bridges reduce the system interconnect fabric logic by reducing the amount of arbitration and multiplexer logic that SOPC Builder generates. This reduction occurs because bridges limit the number of concurrent transfers that can occur. The following sections discuss how you can use bridges to minimize the logic generated by SOPC Builder and optimize system performance.

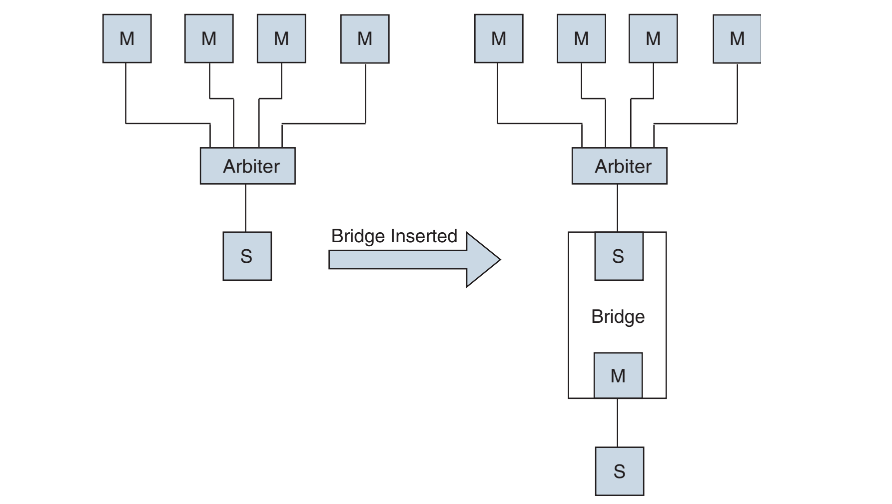
 © June 2008 Altera Corporation

SOPC Builder Speed Optimizations

The system interconnect fabric SOPC Builder generates supports slave-side arbitration. As a result, SOPC Builder creates arbitration logic for every Avalon-MM slave port that is shared by multiple Avalon-MM master ports. SOPC Builder inserts multiplexer logic between master ports that connect to multiple slave ports if both support read datapaths. The amount of logic generated for the system interconnect fabric grows as the system grows.

Even though the interconnect fabric supports multiple concurrent transfers, the master and slave ports in your system can only handle one transfer at a time. If four masters connect to a single slave, the arbiter grants each access in a round robin sequence. If all four masters connect to an Avalon bridge and the bridge masters the slave port, the arbitration logic moves from the slave port to the bridge.

Figure 6–26. Four Masters to Slave Four Masters to Bridge



In Figure 6–26 a pipeline bridge registers the arbitration logic's output signals, including `address` and `writedata`. A multiplexer in the arbitration block drives these signals. Because a logic element (LE) includes both combinational and register logic, this additional pipelining has little or no effect on the logic footprint. And, the additional pipeline stage reduces the amount of logic between registers, increasing system performance.

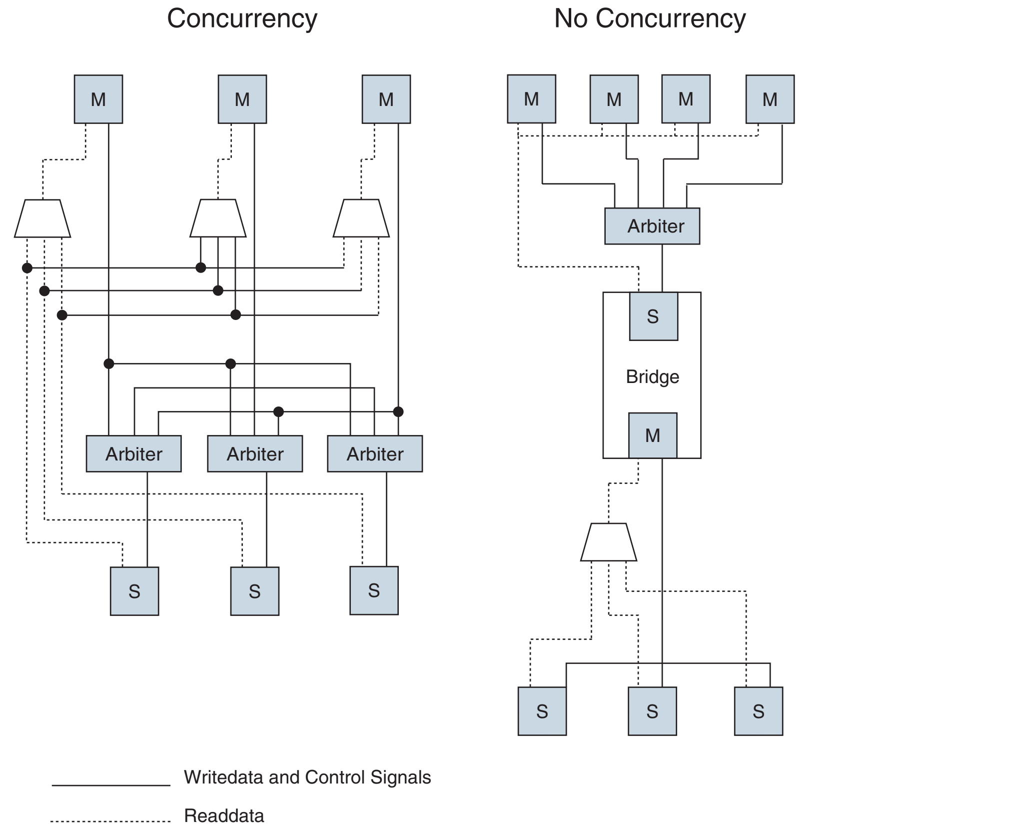
If you can increase the f_{MAX} of your design, you may be able to turn off **Perform register duplication** on the **Physical Synthesis Optimizations** page in the **Settings** dialog box of the Quartus II software. Register duplication duplicates logic in two or more physical locations in the FPGA in an attempt to reduce register-to-register delays. You may also avoid selecting **Speed** for the **Optimization Technique** on the **Analysis & Synthesis Settings** page in the **Settings** dialog box of the Quartus II software. This setting typically results in larger hardware footprint. By making use of the registers or FIFOs available in the Avalon-MM bridges, you can increase the design speed and avoid needless logic duplication or speed optimizations thereby reducing the logic utilization of the design.

Reduced Concurrency

Most embedded designs contain components that are either incapable of supporting high data throughput or simply do not need to be accessed frequently. These components can contain Avalon-MM master or slave ports. Because the system interconnect fabric supports concurrent accesses, you may wish to limit this concurrency by inserting bridges into the datapath to limit the amount of arbitration and multiplexer logic generated. For example, if your system contains three masters and three slave ports that are all interconnected, SOPC Builder generates three arbiters and three multiplexers for the read datapath.

Assuming these masters do not require a significant amount of throughput, you can simply connect all three masters to a pipeline bridge. The bridge masters all three slave ports, effectively reducing the system interconnect fabric into a bus structure. SOPC Builder creates one arbitration block between the bridge and the three masters and single read datapath multiplexer between the bridge and three slaves. This architecture prevents concurrency, just as standard bus structures do. Therefore, this method should not be used for high throughput datapaths. Figure 6–27 illustrates the difference in architecture between system with and without the pipeline bridge.

Figure 6–27. Switch Fabric to Bus



Use Bridges to Minimize Adapter Logic

SOPC Builder generates adapter logic for clock crossing and burst support when there is a mismatch between the clock domains or bursting capabilities of the master and slave port pairs. Burst adapters are created when the maximum burst length of the master is greater than the master burst length of the slave. The adapter logic creates extra logic resources which can be substantial when your system contains Avalon-MM master ports connected to many components that do not share the same characteristics. By placing bridges in your design, you can reduce the amount of adapter logic that SOPC Builder generates.

Effective Placement of Bridges

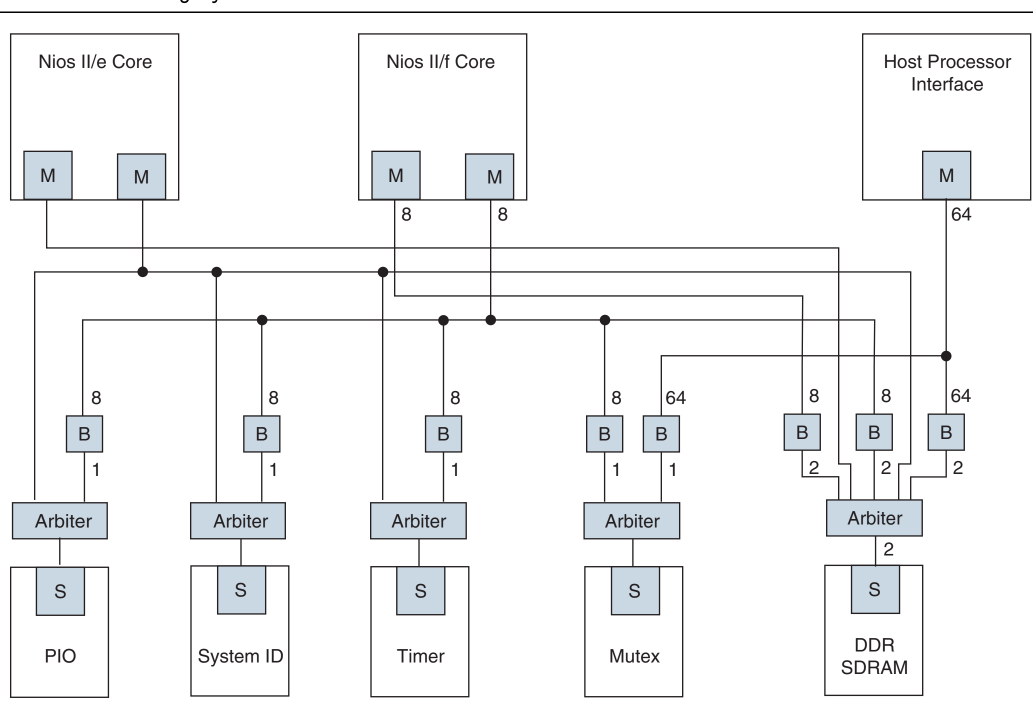
First, analyze each master in your system to determine if the connected slave devices support different bursting capabilities or operate in a different clock domain. The maximum burstcount of a device may be visible as the **burstcount** parameter in the GUI. If it is not, check the width of the `burstcount` signal in the component's HDL file. The maximum burst length is $2^{(width(burstcount -1))}$, so that if the width is 4 bits, the burstcount is 8. If no `burstcount` signal is present, the component does not support bursting or has a burst length of 1.

To determine if a clock crossing adapter is required between the master and slave ports, check the **clock** column beside the master and slave ports in SOPC Builder. If the clock shown is different for the master and slave ports, SOPC Builder inserts a clock crossing adapter between them. To avoid creating multiple adapters, you can place the components containing slave ports behind a bridge so that only one adapter is created. By placing multiple components with the same burst or clock characteristics behind a bridge, you limit concurrency and the number of adapters.

Compact System Example

Figure 6–28 illustrates a system with a mix of components with different burst capabilities. It includes a Nios II/e core, a Nios II/f core and an external processor which offloads some processing tasks to the Nios II/f core. The Nios II/e core maintains communication between the Nios II /f core and external processors. The Nios II/f core supports a maximum burst size of eight. The external processor interface supports a maximum burst length of 64. The Nios II/e core does not support bursting. The only memory in the system is DDR SDRAM with an Avalon maximum burst length of two.

Figure 6–28. Mixed Bursting System



SOPC Builder automatically inserts burst adapters to compensate for burst length mismatches. The adapters reduce bursts to a length of two or single transfers. For the external processor interface connecting to DDR SDRAM, a burst of 64 words is divided into 32 burst transfers, each with a burst length of 2.

At system generation time, SOPC Builder inserts burst adapters based on maximum burstcount values; consequently, the system interconnect fabric includes burst adapters between masters and slave pairs that do not require bursting, if the master is capable of bursts. In Figure 6–28, SOPC Builder inserts a burst adapter between the Nios II processors and the timer, system ID and PIO peripherals. These components do not support bursting and the Nios II processor only performs single word read and write accesses to these devices.

To reduce the number of adapters, you can add pipeline bridges, as Figure 6–29 illustrates. The pipeline bridge between the Nios II/f core and the peripherals that do not support bursts eliminates three burst adapters from Figure 6–28. A second pipeline bridge between the Nios II/f core and the DDR SDRAM, with its maximum burst size set to eight, eliminates another burst adapter.

Figure 6–29. Mixed Bursting System with Bridges

Reducing Power Utilization

Although SOPC Builder does not provide specific features to support low power modes, there are opportunities for you to reduce the power of your system. This section explores the various low power design changes that you can make in order to reduce the power consumption of the system interconnect fabric and your custom components.

Reduce Clock Speeds of Non-Critical Logic

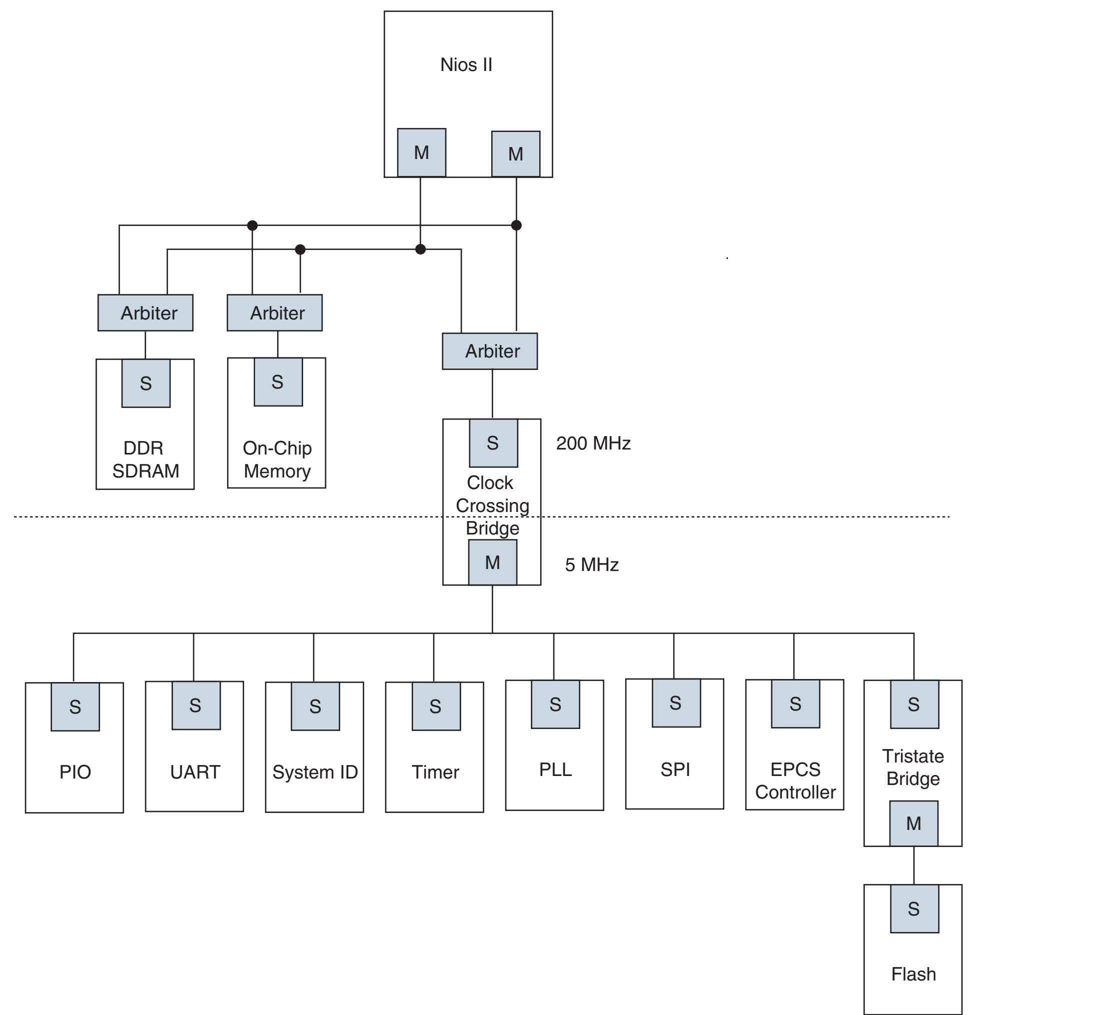
Reducing the clock frequency reduces power consumption. Because SOPC Builder supports clock crossing, you can reduce the clock frequency of the logic that does not require a high frequency clock, allowing you to reduce power consumption. You can use either clock crossing bridges or clock crossing adapters to separate clock domains.

Clock Crossing Bridge

You can use the clock crossing bridge to connect Avalon-MM master ports operating at a higher frequency to slave ports running a a lower frequency. Only low throughput or low priority components should be placed behind a clock crossing bridge that operates at a reduced clock frequency. Examples of typical components that can be effectively placed in a slower clock domain are:

- PIOs
- UARTs (JTAG or RS-232)
- System identification (SysID)
- Timers
- PLL (instantiated within SOPC Builder)
- Serial peripheral interface (SPI)
- EPCS controller
- Tristate bridge and the components connected to the bridge

Figure 6–30. Low Power Using Bridge



Placing these components behind a clock crossing bridge increases the read latency; however, if the component is not part of a critical section of your design the increased latency is not an issue. By reducing the clock frequency of the components connected to the bridge, you reduce the dynamic power consumption of your design. Dynamic power is a function of toggle rates, and decreasing the clock frequency decreases the toggle rate.

Clock Crossing Adapter

SOPC Builder automatically inserts clock crossing adapters between Avalon-MM master and slave ports that operate at different clock frequencies. The clock crossing adapter uses a handshaking state-machine to transfer the data between clock domains. The HDL code that defines the clock crossing adapters resembles that of other SOPC components. Adapters do not appear in the SOPC Builder **Connection** column because you do not insert them. The differences between clock crossing bridges and clock crossing adapters should help you determine which are appropriate for your design.

Throughput

Because the clock crossing bridge uses FIFOs to implement the clock crossing logic, it buffers transfers and data. Clock crossing adapters do not buffer data, so that each transaction is blocking until it completes. Blocking transactions may lower the throughput substantially; consequently, if you wish to reduce power consumption without limiting the throughput significantly you should use the clock crossing bridge. However, if the design simply requires single read transfer, a clock crossing adapter is preferable because the latency is lower than the clock crossing bridge.

Resource Utilization

The clock crossing bridge requires very few logic resources besides on-chip memory. The number of on-chip memory blocks used is proportional to the address span, data width, buffering depth, and bursting capabilities of the bridge. The clock crossing adapter does not use any on-chip memory and requires a moderate number of logic resources. The address span, data width, and bursting capabilities of the clock crossing adapter and also determine the resource utilization of the device.

Throughput versus Memory Tradeoffs

The choice between the clock crossing bridge and clock crossing adapter is between throughput and memory utilization. If on-chip memory resources are limited, you may be forced to choose the clock crossing adapter. Using the clock crossing bridge to reduce the power of a single component may not justify the additional resources required. However, if you can place all your low priority components behind a single clock crossing bridge you reduce power consumption in your design. In contrast, SOPC Builder inserts a clock crossing adapter between each master and slave pair that run at different frequencies if you have not included a clock crossing bridge, increasing the logic utilization in your design.

Minimize Toggle Rates

Your design consumes power whenever logic transitions between on and off states. When the state is held constant between clock edges, no charging or discharging occurs. This section discusses three design techniques you can use to reduce the toggle rates of your system:

- Registering Component Boundaries
- Enabling Clocks
- Inserting Bridges

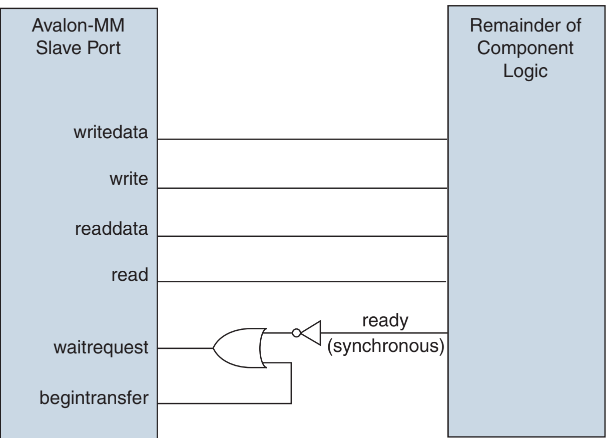
Registering Component Boundaries

The system interconnect fabric is purely combinational when no adapters or bridges are present. When a slave port is not selected by a master, various signals may toggle and propagate into the component. By registering the boundary of your component at the Avalon-MM master or slave interface you can minimize the toggling of the system interconnect fabric and your component. When you register the signals at the port level you must ensure that the component continues to operate within the Avalon-MM specification.

`waitrequest` is usually the most difficult signal to synchronize when you add registers to your component. `waitrequest` must be asserted during the same clock cycle that a master asserts read or write to prolong the transfer. A master interface may read the `waitrequest` signal too early and post more reads and writes prematurely.

For slave interfaces, the system interconnect fabric manages the `begintransfer` signal which is asserted during the first clock cycle of any `read` or `write` transfer. If your `waitrequest` is one clock cycle late you can logically `OR` your `waitrequest` and the `begintransfer` signals to form a new `waitrequest` signal that is properly synchronized.

Figure 6–31. Variable Latency



Or, your component can assert `waitrequest` before it is selected, guaranteeing that the `waitrequest` is already asserted during the first clock cycle of a transfer.

Enabling Clocks

You can use clock enables to hold your logic in a steady state. You can use the `write` and `read` signals as clock enables for Avalon-MM slave components. Even if you add registers to your component boundaries, your interface can still potentially toggle without the use of clock enables.

You can also use the clock enable to disable combinational portions of your component. For example, you can use an active high clock enable to mask the inputs into your combinational logic to prevent it from toggling when the component is inactive. Before preventing inactive logic from toggling, you must determine if the masking causes your circuit to function differently. If this masking causes a functional failure, it may be possible to use a register stage to hold the combinational logic constant between clock cycles.

Inserting Bridges

If you do not wish to modify the component by using boundary registers or clock enables, you can use bridges to help reduce toggle rates. A bridge acts as a repeater where transfers to the slave port are repeated on the master port. If the bridge is not being accessed, the components connected to its master port are also not being accessed. The master port of the bridge remains idle until a master accesses the bridge slave port.

Bridges can also reduce the toggle rates of signals that are inputs to other master ports. These signals are typically `readdata`, `readdatavalid`, and `waitrequest`. Slave ports that support read accesses drive these signals. Using a bridge you can insert either a register or clock crossing FIFO between the slave port and the master to reduce the toggle rate of the master input signals.

Disable Logic

There are typically two types of low power modes: volatile and non-volatile. A volatile low power mode holds the component in a reset state. When the logic is reactivated the previous operational state is lost. A non-volatile low power mode restores the previous operational state. This section covers two ways to disable a component to reduce power using either software- or hardware-controlled sleep modes.

Software Controlled Sleep Mode

To design a component that supports software controlled sleep mode, create a single memory mapped location that enables and disables logic, by writing a 0 or 1. Use the register's output as a clock enable or reset depending on whether the component has non-volatile requirements. The slave port must remain active during sleep mode so that the `enable` bit can be set when the component needs to be activated.

If multiple masters can access a component that supports sleep mode, you can use the mutex core available in SOPC Builder to provide mutual exclusive accesses to your component. You can also build in the logic to re-enable the component on the very first access by any master in your system. If the component requires multiple clock cycles to re-activate then it must assert `waitrequest` to prolong the transfer as it exits sleep mode.

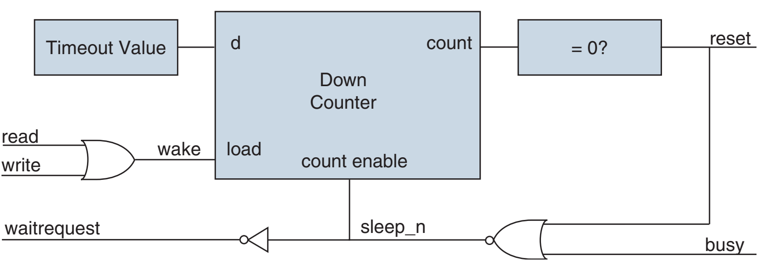
To learn more about the mutex core refer to the *Mutex Core* in volume 5 of the *Quartus II Handbook*.

Hardware Controlled Sleep Mode

You can implement a timer in your component that automatically causes it to enter a sleep mode based upon a timeout value specified in clock cycles between read or write accesses. Each access resets the timer to the timeout value. Each cycle with no accesses decrements the timeout value by 1. If the counter reaches 0, the hardware enters sleep mode until the next access. Figure 6–32 provides a schematic for this logic. If it takes a long time to restore the component to an active state, use a long timeout value so that the component is not continuously entering and exiting sleep mode.

The slave port interface must remain functional while the rest of the component is in sleep mode. When the component exits sleep mode it, must assert the `waitrequest` signal until it is ready for read or write accesses.

Figure 6–32. Hardware Controlled Sleep Components



For more information on reducing power utilization, refer to *Power Optimization in volume 2 of the Quartus II Handbook*.

Referenced Documents

This chapter references the following documents:

- *Accelerated FIR with Built-in Direct Memory Access Example*
- *Avalon Interfaces Specifications*
- *Avalon Memory-Mapped Bridges* chapter in volume 4 of the *Quartus II Handbook*
- *Creating Multiprocessor Nios II Systems Tutorial*
- *Developing Components for SOPC Builder* in volume 4 of the *Quartus II Handbook*
- *Multiprocessor Coordination Peripherals*
- *Mutex Core* in volume 5 of the *Quartus II Handbook*
- *Nios II Embedded Processor Design Examples*
- *Nios II High-Performance Example With Bridges*
- *Power Optimization* in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 6–1 shows the revision history for this chapter.

Table 6–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release	—

7. Memory System Design

ED51008-1.1

Overview

This document describes the efficient use of memories in SOPC Builder 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.

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

Volatile Memory

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

Non-volatile Memory

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

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

On-Chip Memory

On-chip memory is the simplest type of memory for use in an FPGA-based embedded system. The memory is implemented in the FPGA itself; consequently, no external connections are necessary on the circuit board. To implement on-chip memory in your design, simply select the **On-Chip Memory** from the **System Contents** tab in SOPC Builder. 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.

For details about the **On-Chip Memory** SOPC Builder component, refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of *Quartus II Handbook*.

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.

For more information about which types of on-chip memory can be initialized upon FPGA configuration, refer to the *Building Memory Subsystems Using SOPC Builder* chapter of the *Quartus II Handbook*.

Disadvantages

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

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

Best Applications

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

Cache

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

Tightly Coupled Memory

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

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

Look Up Tables

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

FIFO

Embedded systems often need to regulate the flow of data from one system block to another. FIFOs can buffer data between processing blocks that run most efficiently at different speeds. Depending on the size of the FIFO your application requires, on-chip memory can serve as very fast and convenient FIFO storage.

For more information regarding FIFO buffers, refer to the *On-Chip FIFO Memory Core* chapter in volume 5 of the *Quartus II Handbook*.

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.

On-Chip Memory Types

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

Best Practices

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

- Set the on-chip memory data width to match the data-width of its primary system master. For example, if you are connecting the on-chip memory to the data master of a Nios II processor, you should set the data width of the on-chip memory to 32 bits, the same as the data-width of the Nios II data master. Otherwise, the access latency could be longer than one cycle because the system interconnect fabric performs width translation.

- If more than one master connects to an on-chip memory component, consider enabling the dual-port feature of the on-chip memory. The dual-port feature removes the need for arbitration logic when two masters access the same on-chip memory. In addition, dual-ported memory allows concurrent access from both ports, which can dramatically increase efficiency and performance when the memory is accessed by two or more masters. However, writing to both slave ports of the RAM can result in data corruption if there is not careful coordination between the masters.

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

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

External SRAM

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

Advantages

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

See *Building Memory Subsystems Using SOPC Builder* in volume 4 of the *Quartus II Handbook*, for more information regarding shared external buses.

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.

© June 2008 Altera Corporation

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 color conversion algorithm that is too large to fit in on-chip memory.

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

Poor Applications

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

External SRAM Types

There are several types of SRAM devices. The most popular types are listed below.

- 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 turn around cycles, making it a very low-latency. ZBT SRAM typically requires a special controller to take advantage of its low-latency features.

Best Practices

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

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

Flash

Flash memory is a non-volatile memory type used frequently in embedded systems. In FPGA-based embedded systems, flash 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 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 can be written, it must be erased. All flash devices are divided into some number of erase blocks, or sectors, which vary in size, depending on the flash vendor and device size. Entire sections of flash must be erased as a unit; individual words cannot be erased. These requirements sometimes make flash devices difficult to use.

Advantages

The primary advantage of flash memory is that is non-volatile. Modern embedded systems use flash 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 is the most popular for 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.

Refer to *Building Memory Subsystems Using SOPC Builder* in volume 4 of the *Quartus II Handbook* for more information regarding shared external buses.

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, once the write command has been 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 flash.

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

Typical Applications

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

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

Poor Applications

Because of flash memory's slow write speeds, do 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 which use flash memory usually also include some SRAM as well.

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

Flash Types

There are several types of flash devices. The most popular types are listed below:

- CFI flash – This is the most common type of flash memory. It has a parallel interface. CFI stands for common flash interface, a standard to which all CFI flash devices adhere. SOPC Builder and the Nios II processor have built-in support for CFI flash.

  For more details, refer to the following documentation: *Common Flash Interface Controller Core* in volume 5 of the *Quartus II Handbook* and the *Nios II Flash Programmer User Guide.*

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

For more information about EPCS configuration devices, refer to the *Altera Configuration Devices* chapter in volume 2 of Altera's *Configuration Handbook*.

- NAND flash – NAND flash is a newer type of flash memory which has recently begun to gain popularity. NAND flash can achieve very high capacities, up to multiple GBytes per device. The interface to NAND flash is a bit more complicated than that of CFI flash. It requires either a special controller or intelligent low-level driver software. You can use NAND Flash with Altera FPGAs; however, Altera does not provide any built-in support.

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 on the SDRAM controllers available for use in Altera FPGAs, refer to the following documents:

- *DDR and DDR2 SDRAM High-Performance Controller User Guide*,
- *DDR3 SDRAM High-Performance Controller User Guide*
- *AN 398: Using DDR/DDR2 SDRAM with SOPC Builder.*

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.

Refer to *Building Memory Subsystems Using SOPC Builder* in volume 4 of the *Quartus II Handbook* for more information regarding shared external buses.

Disadvantages

Along with the high-capacity and low-cost of SDRAM, comes additional complexity and latency. The complexity of the SDRAM interface requires that you must always use an SDRAM controller to manage SDRAM refresh cycles, address multiplexing, and interface timing. Such a controller consumes FPGA logic elements which 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 the nature of SDRAM dictates that 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 once the initial access latency is overcome because consecutive accesses can be pipelined. Some types of SDRAM can achieve higher clock frequencies than SRAM, further improving throughput. The SDRAM interface specification also employs a burst feature to help improve overall throughput.

Best Applications

SDRAM is generally a good choice in the following circumstances:

- Storing large blocks of data—SDRAM's large capacity makes it the best choice for buffering any 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 reads typical of cache line fills can potentially take advantage of SDRAM's pipeline and burst capabilities.

Poor Applications

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

- Whenever low-latency memory access is required—Although high throughput is possible using SDRAM, its first-access latency is quite high. If low latency access to a particular block of data is a requirement of your application, SDRAM is probably not a good candidate for holding that block of data.

- Small blocks of data—When only a small amount of storage is needed, SDRAM may be unnecessary. An on-chip memory may be able to meet your memory requirements without adding another memory device to the PCB.
- Small, simple embedded systems—If your system uses a small FPGA in which logic resources are scarce and your application does not require the capacity that SDRAM provides, you may prefer to use a small external SRAM or on-chip memory rather than devoting FPGA logic elements to an SDRAM controller.

SDRAM Types

There are a several types of SDRAM devices. The most common types are listed below:

- SDR SDRAM—Single data rate (SDR) SDRAM is the original type of SDRAM. It is either referred to as just SDRAM or 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 a single 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 again improves the potential bandwidth of the memory by improving signal integrity and increasing clock frequencies.

SDRAM Controller Types Available From Altera

Table 7–1 lists the SDRAM controllers that Altera provides. They are available without licenses.

Table 7–1. Memory Controller Available from Altera (Part 1 of 2)

Controller Name	Description
SDR SDRAM Controller	This is the only SDR SDRAM controller Altera offers. It is a simple, easy-to-use controller that works with most available SDR SDRAM devices. For more information refer to *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*.
DDR/DDR2 Controller Megacore Function	This controller is a legacy component which is maintained for existing designs only. Altera does not recommend it for new designs.

Table 7–1. Memory Controller Available from Altera (Part 2 of 2)

Controller Name	Description
High Performance DDR/DDR2 Controller	This is the DDR/DDR2 controller that Altera recommends for new designs. It supports two primary clocking modes, full-rate and half-rate. ■ Full-rate mode presents data to the SOPC Builder system at twice the width of the actual DDR SDRAM device at the full SDRAM clock rate. ■ Half-rate mode presents data to the SOPC Builder system at four times the native SDRAM device data width at half the SDRAM clock rate. For more information about this controller, refer to the *DDR and DDR2 SDRAM High-Performance Controller User Guide*.
High Performance DDR3 Controller	This is the DDR3 controller that Altera recommends for new designs. It is similar to the high performance DDR/DDR2 controller. It also supports full- and half-rate clocking modes. For more information about this controller, refer to the *DDR3 SDRAM High-Performance Controller User Guide*.

Best Practices

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

Half-Rate Mode

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

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

Full-Rate Mode

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

When using full-rate mode, the local data width of the SDRAM controller is two times the data width of the physical DDR SDRAM itself. 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. Again, this choice facilitate bursts to the SDRAM device

Sequential Access

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

Bursting

SDRAM devices employ bursting to improve throughput. Bursts group a number of transactions to sequential addresses, allowing data to be transferred back-to-back without the 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 see if bursting is supported.

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

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

SDRAM Minimum Frequency

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

SDRAM Device Speed

SDRAM devices, both SDR and DDR, come in several speed grades. When using SDRAM with FPGAs, the operating frequency of the FPGA system is usually lower than the maximum capability of the SDRAM device. Therefore, it is typically not worth the extra cost to use fast speed-grade SDRAM devices. Before committing to a specific SDRAM device, consider both the expected SDRAM frequency of your system, and the maximum and minimum operating frequency of the particular SDRAM device.

Memory Optimization

This section presents tips and tricks that can be helpful when implementing any type of memory in your SOPC Builder system. These techniques can help improve system performance and efficiency.

Isolate Critical Memory Connections

For many systems, particularly complex ones, isolating performance-critical memory connections is beneficial. To achieve the maximum throughput potential from memory, connect it to the fewest number of masters possible and share those masters with the fewest number of slaves possible. Minimizing connections reduces the size of the data multiplexers required, increasing potential clock speed and also reduces the amount of arbitration necessary to access the memory.

You can use bridges to isolate memory connections. For more information on efficient system topology refer to the following documents:

- *Avalon Memory-Mapped Bridges* chapter in volume 4 of the *Quartus II Handbook.*
- *Avalon Memory-Mapped Design Optimizations* chapter of the *Embedded Design Handbook*.

Match Master and Slave Data Width

Matching the data widths of master and slave pairs in SOPC Builder is advantageous. Whenever a master port is connected to a slave of a different data width, SOPC Builder 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 CPU performance is critical to your overall system performance, and the CPU 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 CPU 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 these cases, it may be advantageous to implement a 64-bit SDRAM interface.

Use Separate Memories to Exploit Concurrency

Any time multiple masters in your system access the same memory, each master is only granted access 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 where the potential for parallelism is significant.

In SOPC Builder, 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 SOPC Builder. 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.

For more information regarding topology tradeoffs refer to *Avalon Memory-Mapped Design Optimizations* chapter of the *Embedded Design Handbook*.

Understand the Nios II Instruction Master Address Space

This Nios II CPU 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 that can address up to 2 GBytes.

Test Memory

You should rigorously test the memory in your system to ensure that it is physically connected and setup properly before relying on it in an actual application. The Nios II Development Kit ships with a memory test example which is a good starting point for building a thorough memory test for your system.

Case Study

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

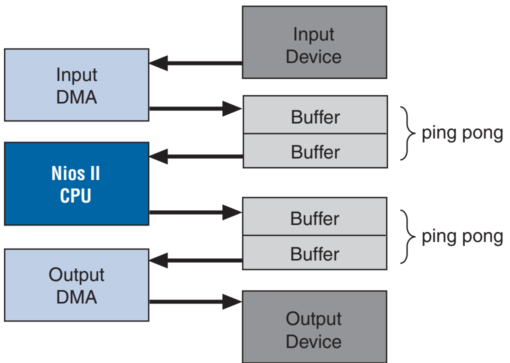
Application Description

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 into a buffer.
2. A Nios II CPU 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 CPU 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 CPU is processing the current buffer, a technique commonly called ping-ponging.

Figure 7–1 shows the basic architecture of the system.

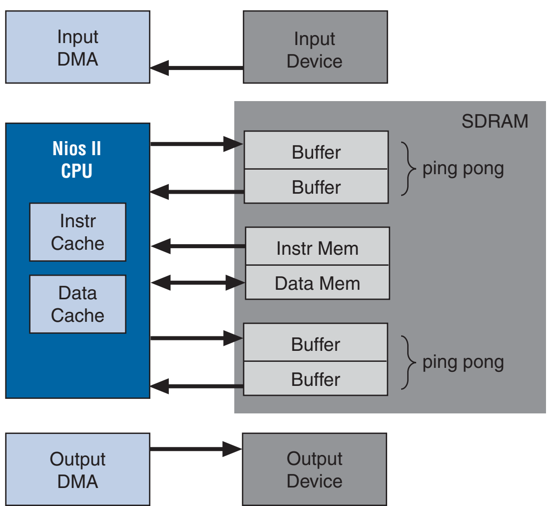
Figure 7–1. Sample Application Architecture



Initial Memory Partitioning

As a starting point, the application uses SDRAM for all of its storage and buffering, a commonly used memory architecture. The input DMA copies data from the video source to an input buffer in SDRAM. The CPU performs its processing by reading from that SDRAM input buffer, writing its result to an output buffer, also located in SDRAM. In addition, the CPU uses SDRAM for both its instruction and data memory. (Refer to Figure 7–2.)

Figure 7–2. 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 Figure 7–2 illustrates, there are 6 different channels of data being accessed in the SDRAM.

1. CPU instruction
2. CPU data
3. Input data from DMA

4. Input data to CPU
5. Output data from CPU
6. Output data to DMA

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

Optimized Memory Partitioning

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

Add An External SRAM for input buffers

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

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

The redesigned system processes data faster, at the expense of more complexity and higher cost. Figure 7–3 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 7–3. Input Channel Moved to External SSRAM

Notice that there are still four channels connected to SDRAM:

1. CPU instruction
2. CPU data
3. Output data from CPU
4. Output data to DMA

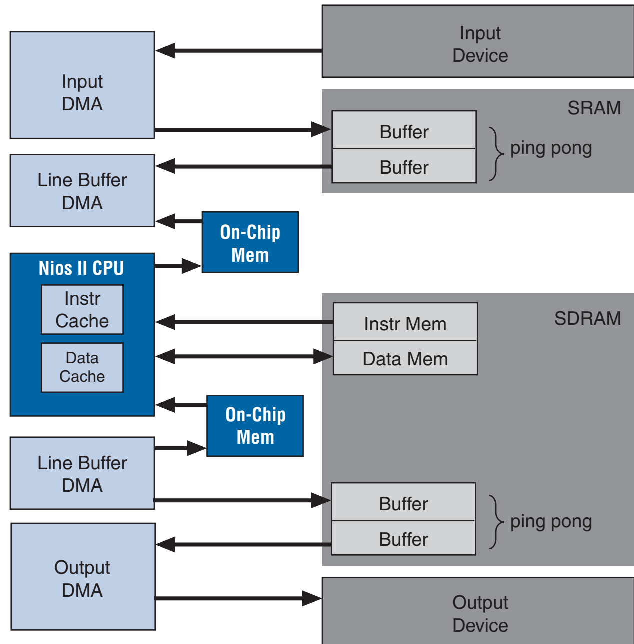
While we could probably achieve some additional performance benefit by adding a second external SRAM for the output channel, the benefit would not likely be significant enough to outweigh the added cost and complexity. The reason is that only two of the four remaining channels require significant bandwidth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

Add On-Chip Memory for Video Line Buffers

The final optimization is to add small on-chip memory buffers for input and output video lines. Because the processing algorithm operates on the video input one line at a time, buffering entire lines of input data in an on-chip memory improves performance. It enables the CPU 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 CPU 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 CPU output. The CPU writes its output data to an on-chip memory line buffer, which is copied to the output frame buffer by a DMA once both the input and output ping-pong buffers flip, and the CPU begins processing the next line. Figure 7–4 illustrates this memory architecture.

Figure 7–4. On-Chip Memories Added As Line Buffers



Referenced Documents

This chapter references the following documents:

- *Altera Configuration Devices* chapter in volume 2 of the *Configuration Handbook*
- *AN 398: Using DDR/DDR2 SDRAM with SOPC Builder*
- *Avalon Memory-Mapped Bridges* in volume 4 of the *Quartus II Handbook*
- *Avalon Memory-Mapped Design Optimizations* chapter of the *Embedded Design Handbook*
- *Building Memory Subsystems Using SOPC Builder* in volume 4 of *Quartus II Handbook*
- *Common Flash Interface Controller Core* in volume 5 of the *Quartus II Handbook*
- *DDR and DDR2 SDRAM High-Performance Controller User Guide*
- *DDR3 SDRAM High-Performance Controller User Guide*
- *Nios II Flash Programmer User Guide*
- *On-Chip FIFO Memory Core* in volume 5 of the *Quartus II Handbook*
- *SDRAM Controller Core* in volume 5 of the *Quartus II Handbook*
- *Using Nios II Tightly Coupled Memory Tutorial*

© June 2008 Altera Corporation

Document Revision History

Table 7–1 shows the revision history for this chapter.

Table 7–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release.	—

8. Hardware Acceleration and Coprocessing

ED51006-1.1

This chapter discusses how you can use hardware accelerators and coprocessing to create more efficient, higher throughput designs in SOPC Builder. This chapter discusses the following topics:

- Accelerating Cyclic Redundancy Checking (CRC)
- Creating Nios II Custom Instructions
- Using the C2H Compiler
- Creating Multicore Designs
- Pre- and Post-Processing
- Replacing State Machines

Hardware Acceleration

Hardware accelerators implemented in FPGAs offer a scalable solution for performance-limited systems. Other alternatives for increasing system performance include choosing higher performance components or increasing the system clock frequency. Although these other solutions are effective, in many cases they lead to additional cost, power, or design time.

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. Figure 8–1 illustrates a system in which a Nios® II processor offloads CRC processing to a hardware accelerator. In this system, the Nios II processor reads and writes registers to control the CRC using its Avalon® Memory-Mapped (Avalon-MM) slave port. The CRC component's Avalon-MM master port reads data for the CRC check from memory.

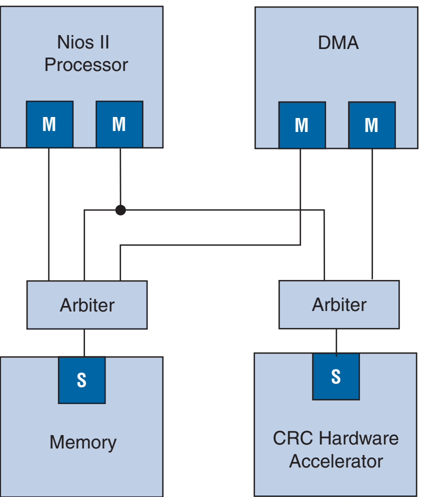
This design example and the HDL files to implement it are fully explained in the *Developing Components for SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.

Figure 8–1. A Hardware Accelerator for CRC

Nios II Processor
M M
CRC Hardware Accelerator
S M
Arbiter
S
Memory

An alternative approach includes a dedicated DMA engine in addition to the Nios II processor. Figure 8–2 illustrates this design. In this system, the Nios II processor programs the DMA engine which transfers data from memory to the CRC.

Figure 8–2. DMA and Hardware Accelerator for CRC



Although Figure 8–2 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 SOPC Builder system using the component editor.

To learn more about using component editor, refer to the *Component Editor* in volume 4 of the *Quartus II Handbook*. You can find additional examples of hardware acceleration on Altera's Hardware Acceleration web page.

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.

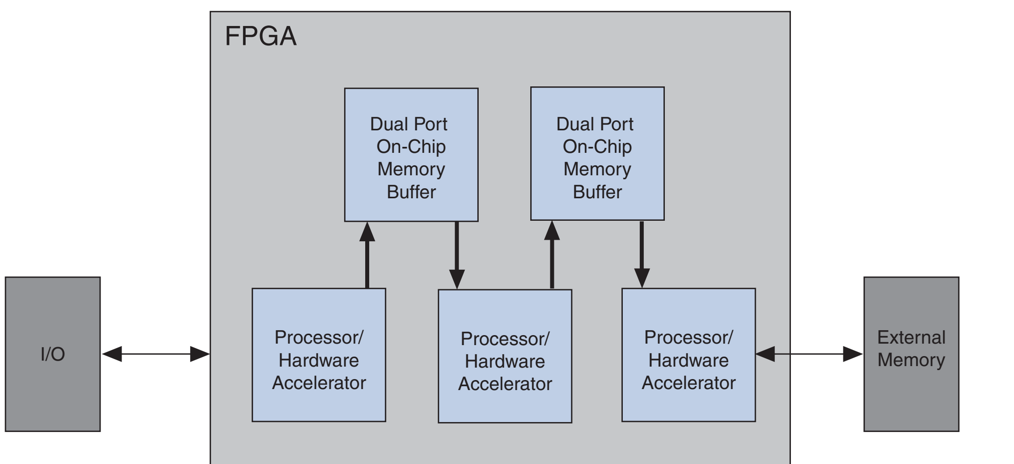
Typically, memory interfaces cause the most problems in systems that contain multiple processors and hardware accelerators. The following recommendations on interface design can maximize the throughput of your hardware accelerator:

- Match high performance memory and interfaces to the highest priority tasks your system must perform.
- Give high priority tasks a greater share of the I/O bandwidth if any memory or interface is shared.
- If you have multiple processors in your system, but only one of the processors provides real-time functionality, assign it a higher arbitration share.

Pipelining Algorithms

A common problem in systems with multiple Avalon-MM master ports is competition for shared resources. You can improve performance by pipelining the algorithm and buffering the intermediate results in separate on-chip memories. Figure 8–3 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 8–3. Using On-Chip Memory to Achieve High Performance



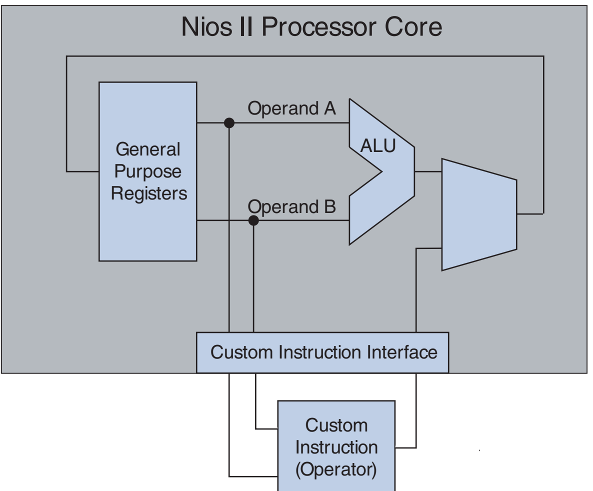
To learn more about the topics discussed in this section refer to the following documentation: *System Interconnect Fabric for Memory-Mapped Interfaces* in volume 4 of the *Quartus II Handbook* and *Building Memory Subsystems Using SOPC Builder* in volume 4 of the *Quartus II Handbook*. To learn more about optimizing memory design refer to *Memory System Design* in volume 3 of the *Embedded Design Handbook.*

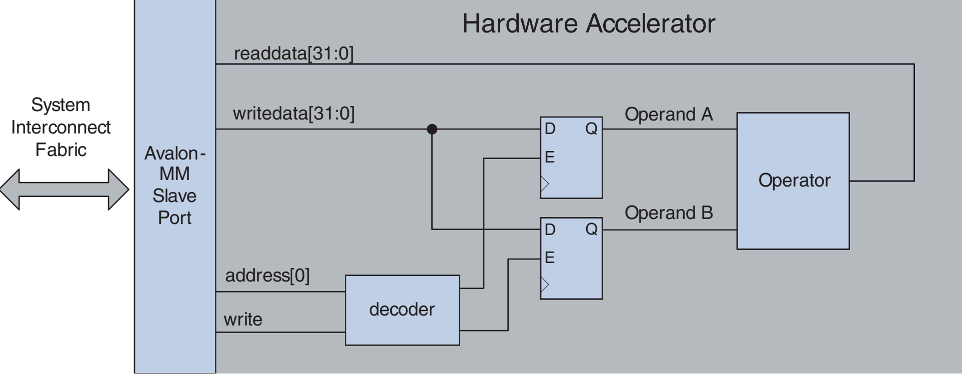
Creating Nios II Custom Instructions

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 the same structure. They include up to two inputs and one output. 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. Figure 8–4 illustrates the differences in implementation between a custom instruction and a hardware accelerator.

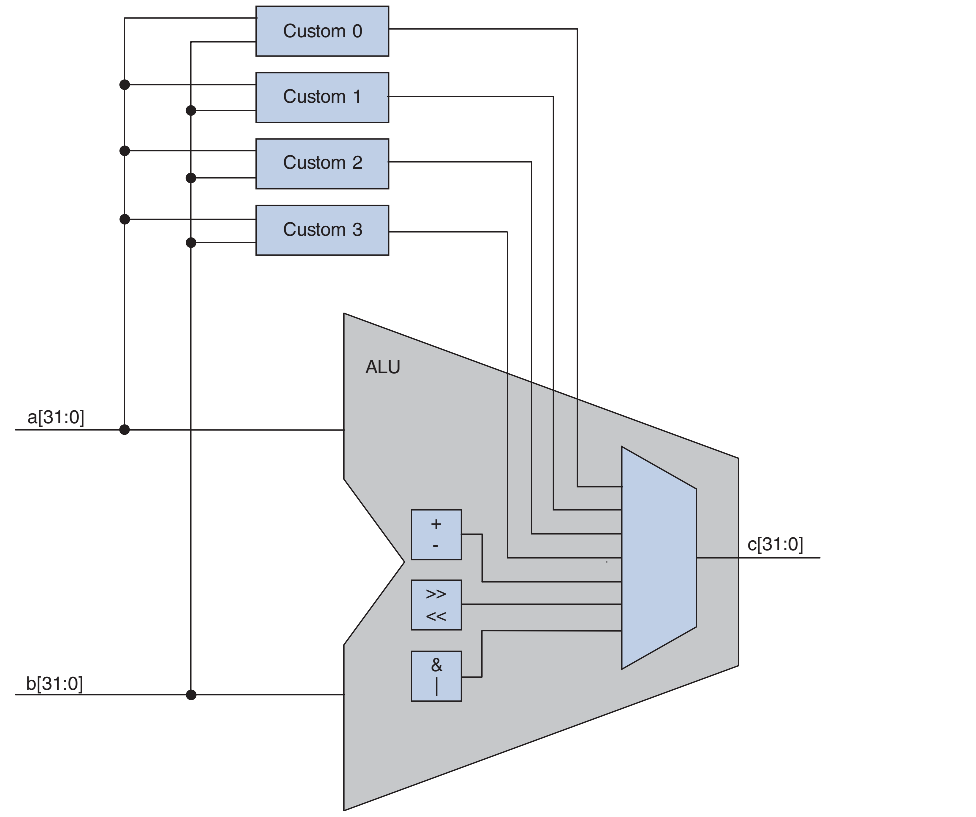
Figure 8–4. Implementation Differences between a Custom Instruction and Hardware Accelerator





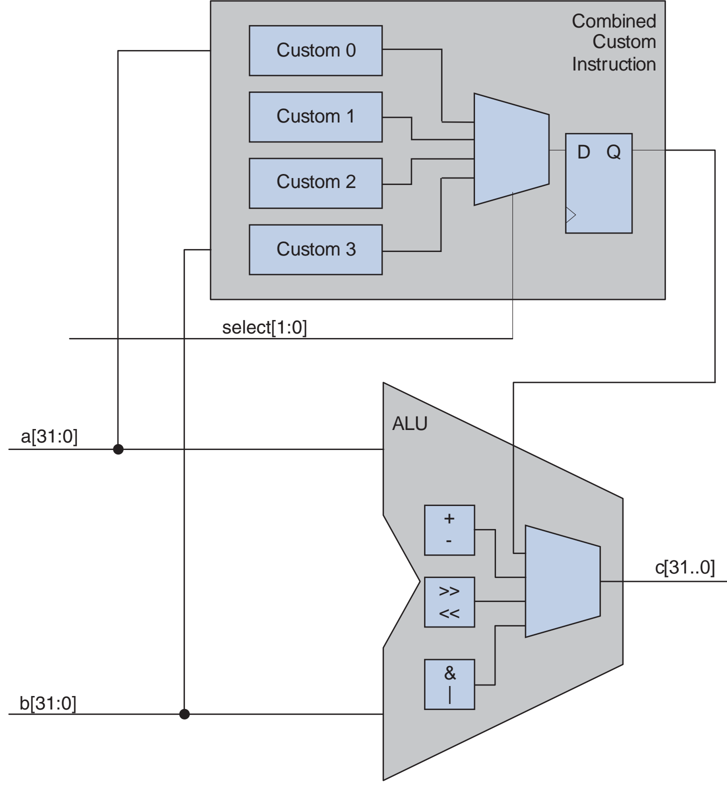
Because custom instructions extend the Nios II processor's ALU, the logic must meet timing or the f_{MAX} of the processor will suffer. As you add custom instructions to the processor, the ALU multiplexer grows in width as Figure 8–5 illustrates. This multiplexer selects the output from the ALU hardware (`c[31:0]` in Figure 8–5). 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 8–5. Individual Custom Instructions



Instead of adding several custom instructions, you can combine the functionality into a single logic block as shown in Figure 8–6. 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 8–6. Combined Custom Instruction



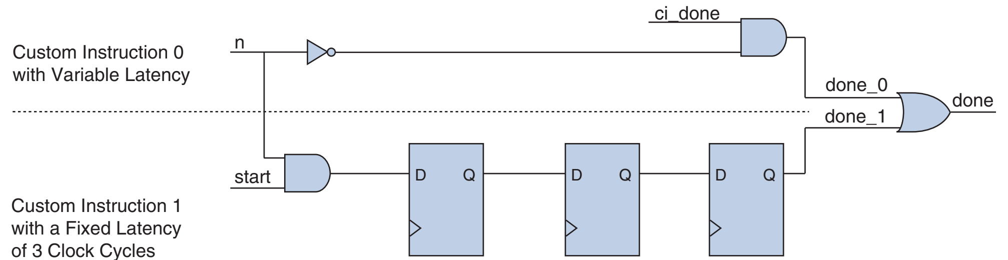
Notes to Figure 8–6:

(1) The Nios II compiler calls the select wires to the multiplexer <*n*>.

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. Figure 8–7 shows the simplest method to implement this conversion by shifting the `start` bit by <*n*> clock cycles and logically `OR`ing all the `done` bits.

Figure 8–7. Sequencing Logic for Mixed Latency Combined Custom Instruction



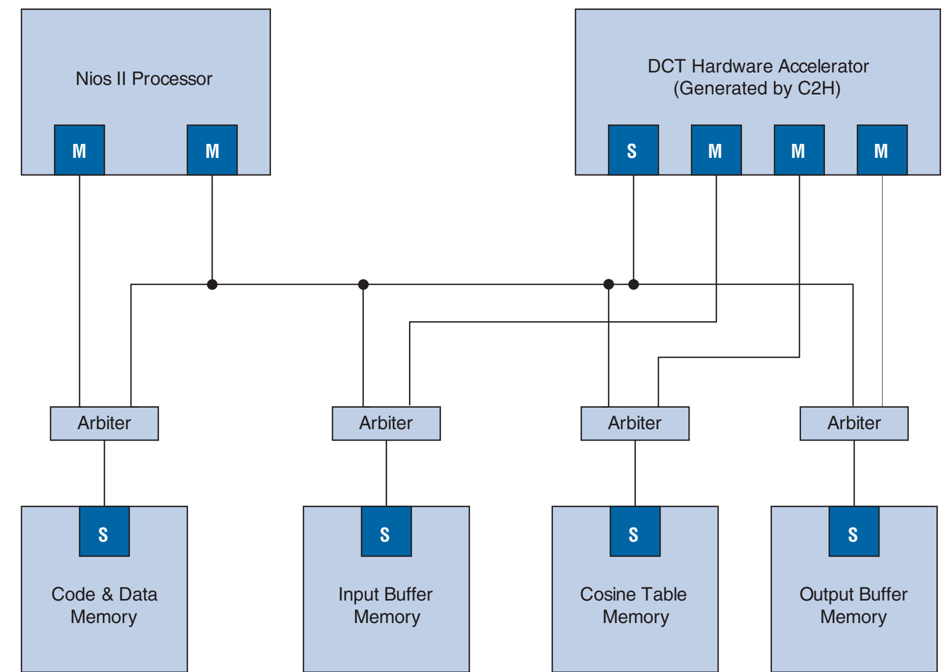
For more information about creating and using custom instructions see the *Nios II Custom Instruction User Guide*.

Using the C2H Compiler

You can use the Nios II C2H Compiler to compile your C source code into HDL synthesizable source code. SOPC Builder automatically places your hardware accelerator into your system. SOPC Builder automatically connects all the master ports to the necessary memories and connects the Nios II processor data master to the accelerator slave port which is used to transfer data.

Choose the Nios II C2H Compiler instead of custom instructions when your algorithm requires access to memory. The C2H Compiler creates Avalon-MM masters that access memory. If your algorithm accesses several memories, the C2H Compiler creates a master per memory access, allowing you to benefit from the concurrent access feature of the system interconnect fabric. You can also use the C2H Compiler to create hardware accelerators that are non-blocking so that you can use the accelerator in parallel with other software functionality.

Figure 8–8. C2H Discrete Cosine Transform (DCT) Block Diagram



In figure Figure 8–8 the two-dimensional DCT algorithm is accelerated to offload a Nios II processor. The DCT algorithm requires access to input and output buffers as well as a cosine lookup table. Assuming that each resides in separate memories, the hardware accelerator can access all three memories concurrently.

For more information please refer to the *Nios II C2H Compiler User Guide* and the *Optimizing C2H Compiler Results* chapter in the *Embedded Design Handbook*. There are also C2H examples available on the Altera website.

Coprocessing

Partitioning system functionality between a Nios II processor and hardware accelerators or between multiple Nios II processors in your FPGA can help you control costs. The following sections demonstrate how you can use coprocessing to create high performance systems.

Creating Multicore Designs

Multicore designs combine multiple processor cores in a single FPGA to create a higher performance computing system. Typically, the processors in a multicore design can communicate with each other. Designs including the Nios II processor can implement inter-processor communication, or the processors can operate autonomously.

When a design includes more than one processor you must partition the algorithm carefully to make efficient use of all of the processors. The following example includes a Nios II-based system that performs video over IP, using a network interface to supply data to a discrete DSP processor. The original design overutilizes the Nios II processor. The system performs the following steps to transfer data from the network to the DSP processor:

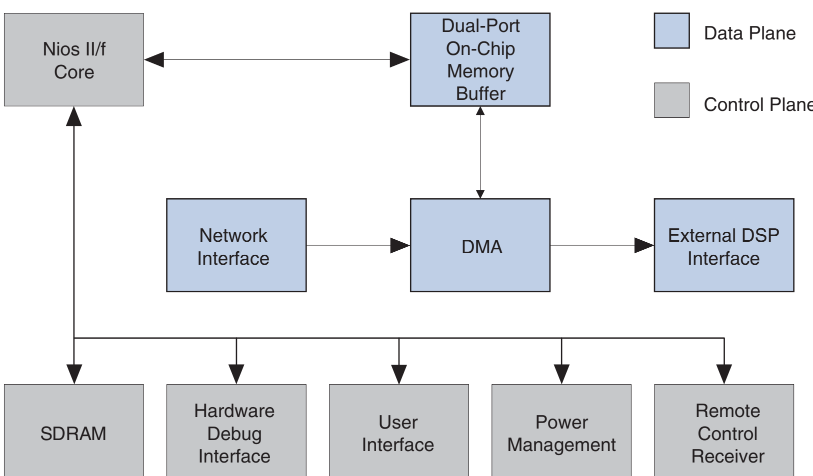
1. The network interface signals when a full data packet has been received.
2. The Nios II processor uses a DMA engine to transfer the packet to a dual-port on-chip memory buffer.
3. The Nios II processor processes the packet in the on-chip memory buffer.
4. The Nios II processor uses the DMA engine to transfer the video data to the DSP processor.

In the original design, the Nios II processor is also responsible for communications with the following peripherals that include Avalon-MM slave ports:

- Hardware debug interface
- User interface
- Power management
- Remote control receiver

Figure 8–9 illustrates this design.

Figure 8–9. Over-utilized Video System

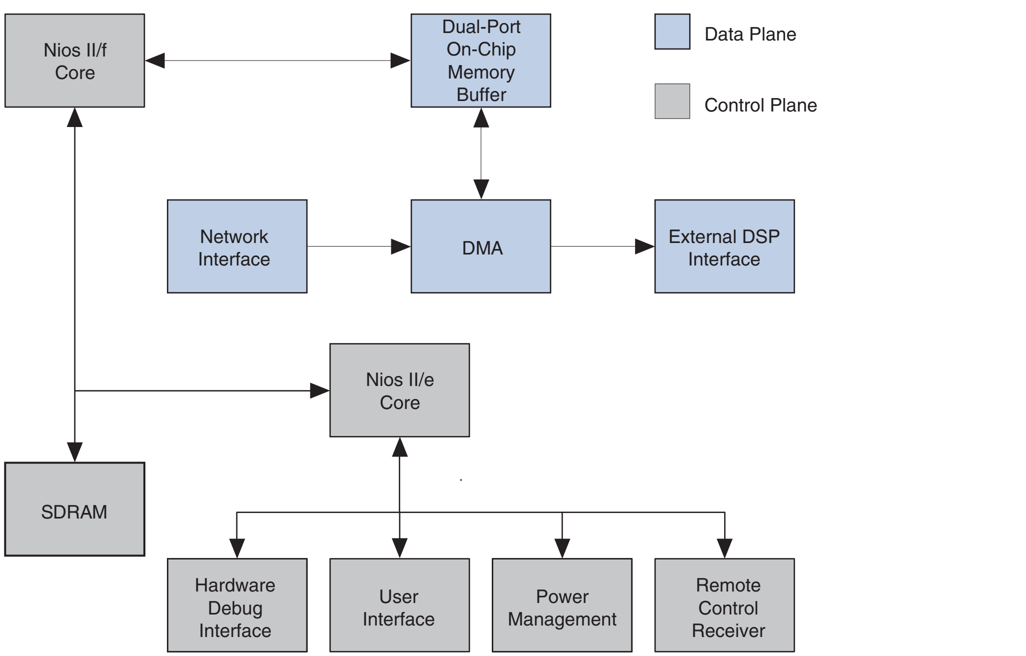


Adding a second Nios II processor to the system, allows the workload to be divided so that one processor handles the network functionality and the other the control interfaces. Figure 8–10 illustrates the revised design.

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 on designing hardware and software for inter-processor communication, refer to the *Creating Multiprocessor Nios II Systems Tutorial* and *Multiprocessor Coordination Peripherals* in volume 5 of the *Quartus II Handbook.* Refer to the *Nios II Processor Reference Handbook* for complete information on this soft core processor. A Nios II Multiprocessor Design Example is available on the Altera website.

Figure 8–10. High Performance Video System



In Figure 8–10, the second Nios II processor added to the system performs primarily low-level maintenance tasks; consequently, the Nios II/e core is used. The Nios II/e core implements only the most basic processor functionality in an effort to trade off performance for a small hardware footprint. This core is approximately one-third the size of the Nios II/f core.

To learn more about the three Nios II processor cores refer to the *Nios II Core Implementation Details* chapter in the *Nios II Processor Reference Handbook*.

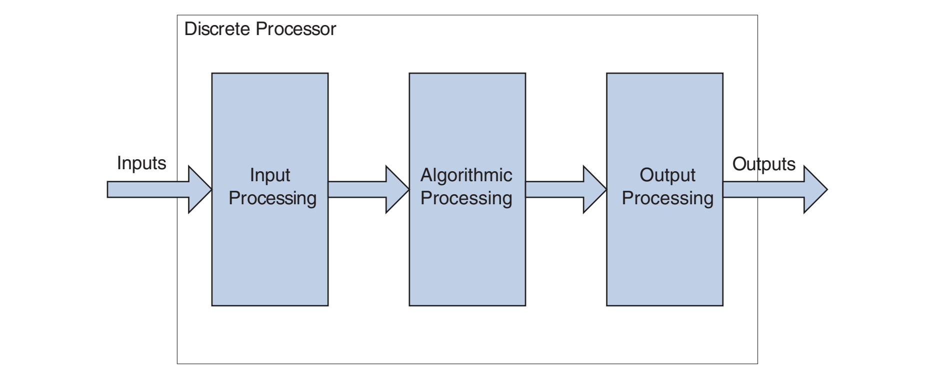
Pre- and Post-Processing

The high performance video system illustrated in Figure 8–10 distributes the workload by separating the control and data planes in the hardware. Figure 8–11 illustrates a different approach. All three stages of a DSP workload are implemented in software running on a discrete processor. This workload includes the following stages:

- Input processing—typically removing packet headers and error correction information
- Algorithmic processing and error correction—processing the data
- Output processing—typically adding error correction, converting data stream to packets, driving data to I/O devices

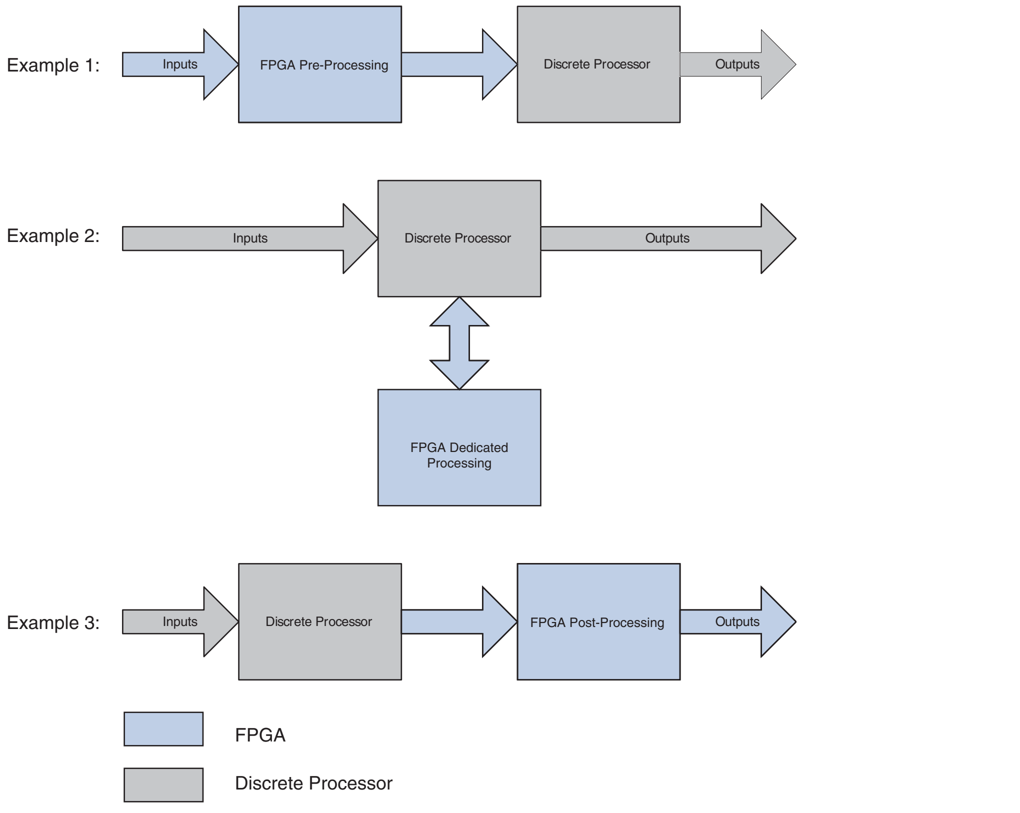
By off loading the processing required for the inputs or outputs to an FPGA, the discrete processor has more computation bandwidth available for the algorithmic processing.

Figure 8–11. Discrete Processing Stages



If the discrete processor requires more computational bandwidth for the algorithm, dedicated coprocessing can be added. Figure 8–12 below shows examples of dedicated coprocessing at each stage.

Figure 8–12. Pre- Dedicated, and Post-Processing



Replacing State Machines

You can use the Nios II processor to implement scalable and efficient state machines. When you use dedicated hardware to implement state machines, each additional state or state transition increases the hardware utilization. In contrast, adding the same functionality to a state machine that runs on the Nios II processor only increases the memory utilization of the Nios II processor.

A key benefit of using Nios II for state machine implementation is the reduction of complexity that results from using software instead of hardware. A processor, by definition, is a state machine that contains many states. These states can be stored in either the processor register set or the memory available to the processor; consequently, state machines that would not fit in the footprint of a FPGA can be created using memory connected to the Nios II processor.

When designing state machines to run on the Nios II processor, you must understand the necessary throughput requirements of your system. Typically, a state machine is comprised of decisions (transitions) and actions (outputs) based on stimuli (inputs). The processor you have chosen determines the speed at which these operations take place. The state machine speed also depends on the complexity of the algorithm being implemented. You can subdivide the algorithm into separate state machines using software modularity or even multiple Nios II processor cores that interact together.

Low-Speed State Machines

Low-speed state machines are typically used to control peripherals. The Nios II/e processor pictured in Figure 8–10 on page 8–10 could implement a low speed state machine to control the peripherals.

Even though the Nios II/e core does not include a data cache, Altera recommends that the software accessing the peripherals use data cache bypassing. Doing so avoids potential cache coherency issues if the software is ever run on a Nios II/f core that includes a data cache.

For information regarding data cache bypass methods, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

State machines implemented in SOPC Builder 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 SOPC Builder are no different than those you would use if you were creating a state machine manually. One noticeable difference in the SOPC Builder 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 on the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

High-Speed State Machines

You should implement high throughput state machine using a Nios II/f core. To maximize performance, focus on the I/O interfaces and memory types. The following recommendations on memory usage can maximize the throughput of your state machine:

- Use on-chip memory to store logic for high-speed decision making.
- Use tightly-coupled memory if the state machine must operate with deterministic latency. Tightly-coupled memory has the same access time as cache memory; consequently, you can avoid using cache memory and the cache coherency problems that might result.

Refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook* for more information on tightly-coupled memory.

Subdivided State Machines

Subdividing a hardware-based state machine into smaller more manageable units can be difficult. If you choose to keep some of the state machine functionality in a hardware implementation, you can use the Nios II processor to assist it. For example, you may wish to use a hardware state machine for the high data throughput functionality and Nios II for the slower operations. If you have partitioned a complicated state machine into smaller, hardware based state machines, you can use the Nios II processor for scheduling.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Building Memory Subsystems Using SOPC Builder* in volume 4 of the *Quartus II Handbook*
- *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*
- *Component Editor* in volume 4 of the *Quartus II Handbook*
- *Creating Multiprocessor Nios II Systems Tutorial*
- *Developing Components for SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*
- *Memory Systedm Design* in volume 3 of the *Embedded Design Handbook*
- *Nios II C2H Compiler User Guide*
- *Nios II Custom Instruction User Guide*
- *Optimizing C2H Compiler Results* chapter in the *Embedded Design Handbook*
- *Multiprocessor Coordination Peripherals* in volume 5 of the *Quartus II Handbook*
- *Nios II Core Implementation Details* chapter in the *Nios II Processor Reference Handbook*
- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*

- *System Interconnect Fabric for Memory-Mapped Interfaces* in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 8–1 shows the revision history for this chapter.

Table 8–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release	—

9. Verification and Board Bring-Up

ED51010-1.2

Introduction

This chapter provides an overview of the tools available in the Quartus® II software and the Nios® II Embedded Design Suite (EDS) that you can use to verify and bring up your embedded system.

This chapter covers the following topics:

- Verification Methods
- Board Bring-up
- System Verification

Verification Methods

Embedded systems can be difficult to debug because they have limited memory and I/O and consist of a mixture of hardware and software components. Altera® provides the following tools and strategies to help you overcome these difficulties:

- FS2 Console
- System Console
- SignalTap II Embedded Logic Analyzer
- External Instrumentation
- Stimuli Generation

Prerequisites

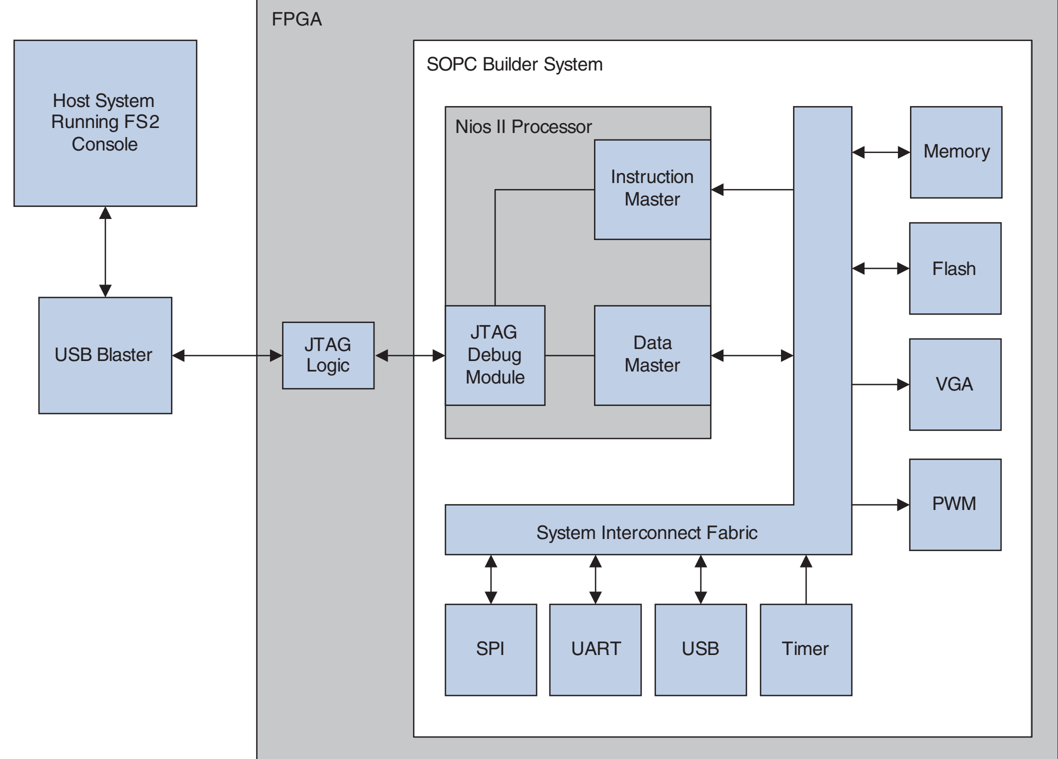
To make effective use of this chapter, you should be familiar with the following topics:

- Defining and generating Nios II hardware systems with SOPC Builder
- Compiling Nios II hardware systems with the Quartus II development software

FS2 Console

The FS2 console, developed by First Silicon Solutions (FS2) extends the verification functionality of the Nios II processor. The FS2 console communicates with the Nios II JTAG debug module that is available for all three variants of the Nios II processor. FS2 optionally uses an external system analyzer hardware module that creates additional trace support to the Nios II JTAG debug module. Figure 9–1 illustrates the connectivity between an FS2 console and an SOPC Builder system.

Figure 9–1. FS2 Console Communication Path



The Nios II JTAG debug module uses the Nios II data master port to communicate with components that contain Avalon® Memory-Mapped (Avalon-MM) slave ports. Although the Nios II JTAG debug module is tightly integrated with the Nios II processor, it does not rely on any additional support being provided by the processor. As a result, you can use the Nios II JTAG debug module and the FS2 console to verify a system without having to write software.

The FS2 console does not support host machines running the Linux operating system.

SOPC Builder Test Integration

Even if you do not intend to include a Nios II processor or an SOPC Builder system in your final design, you can still include a Nios II processor during the debug phase to take advantage of the embedded tools that Altera provides. The Nios II processor contains a data master which you can use to perform read and write accesses to your hardware blocks.

To include a JTAG debug module in your system follow these steps:

1. On the **System Contents** tab, double-click the **Nios II Processor** component.
2. In the Nios II Processor wizard, click the **JTAG Debug Module** tab.
3. Make sure **Level 1** is selected.

The JTAG debug module is required for communication between your system and the FS2 console.

Capabilities of the FS2 Console

You can launch the F2S console from within the Nios II IDE or from the Nios II command shell. Once the FS2 console is open, you have access to the command line and scripting capabilities of the software. The command line within the FS2 console is sufficient for lightweight debugging. To access help for FS2, simply type `help` for a list of available commands. The help system is hierarchical. When you type `help`, the help system lists the top-level command hierarchy. You can refine your help searching by typing `help <command_name>` to learn more about the commands available. For example, if you type `help memory` The FS2 console displays a list all of the commands to access memory, including: **addr**, **asm**, **byte**, **compare**, **copy, dasm**, **dump**, and so on.

Using the FS2 console you can query the FPGA to determine if there are any Nios II debug modules present. The FS2 console can access a Nios II debug module anywhere on the JTAG chain. Because your design may have multiple Nios II debug modules, you can specify the debug module you prefer.

The Nios II processor has a 32-bit data master. Using the FS2 console, you can perform either byte (`byte`), half word (`half`), or word (`word`) accesses to any Avalon-MM slave port.

The FS2 console supports the Tcl/Tk scripting language. Scripting memory accesses is particularly useful if you have many hardware blocks to test or need to instrument regression testing. A Tcl/Tk reference guide is integrated into the FS2 console help menu.

sld info Command

The `sld info` command lists the JTAG chains that are available on your board. For the chain that it is being used to access your board, this command provides identifying information for the JTAG cable (hw), FPGA device or devices (device) and debug modules (node). Figure 9–2 shows typical output from this command. In this example, communication occurs over the second JTAG chain, Hw 1: USB-Blaster [USB-0]. There is a single FPGA in this JTAG chain, device 1: EP2C35, and there is a single debug module on this chain, node 0: owner: First Silicon Solutions.

You must specify these components to the FS2 Console using the `config` command. For the JTAG chain illustrated in Figure 9–2, you must type the following three commands:

- `config sldHW 1`—selects the second programming cable
- `config sldDev 1`—selects the second device on the JTAG chain

- `config sldNode 0`—selects the first debug module in the FPGA

Figure 9–2. sld info Command

```
13> sld info
aji hw 0: ByteBlaster [LPT1]
      Can't Lock hardware[0]'s device!
aji hw 1: USB-Blaster [USB-0]
   aji device 1: EP2C35
       aji node 0: owner: First Silicon Solutions; ver: 3; Id: 22
       aji node -: owner:      Altera Corporation; ver: 1; Id: 80; Instance: 0

14>
```

You can update this configuration information to communicate over a different JTAG cable to a different device and debug module. For example, to communicate over the fourth programming cable to the third device using the second Nios II debug module, you would type the following commands:

- `config sldHW 3`
- `config sldDev 2`
- `config sldNode 1`

When you first bring up the FS2 Console, it is initialized to communicate over the first cable, to the first device and using the debug module in the first FPGA. However, if you update this information, your changes are persistent.

After you compile your design with the Quartus II software, the JTAG debug interface file (**.jdi)** in your project directory includes the debug module instance numbers. If your design includes two SOPC Builder systems in a single FPGA, the debug module instance numbers may change when you recompile. Each debug module is referenced by its full name and level of hierarchy in the design. The debug module number is stored as `sld_instance_index`. For example, if a debug module is assigned to three, the **.jdi** file includes the following setting:

```
<parameter name="sld_instance_index" type="dec" value="3"/>
```

This is the value you use when setting sldNode.

FS2 Examples

The following procedure writes 0x5A followed by 0xA5 to an 8-bit hardware block:

1. Download the hardware image file SRAM object file (**.sof**).
2. In the Nios II command shell, type `nios2-console` to start the FS2 console.
3. In the FS2 console, type `openport sld` to establish communication with a remote debugger.
4. Type `halt` to stop the Nios II processor.
5. Type `byte 0x00810880 0x5A` to write 0x5A to memory location 0x00810880.
6. Type `byte 0x00810880 0xA5` to write 0xA5 to memory location 0x00810880.

Example 9–1 writes a repeating pattern to an address range. Before trying this example, check the Base (address) column for a memory device in your SOPC Builder system, so that you write and read valid locations. In this example, an on-chip memory has a base address of 0x02100000.

Example 9–1. Writing a Repeating Pattern to an Address Range

```
# write a repeating pattern of 0x5a5a5a5a to an address range
word 0x02100000..0x021000FF 0x5a5a5a5a
# read back the data from the address range
dump 0x02100000..0x021000FF word
```

All the commands sent to the JTAG debug module from the FS2 console use the JTAG interface of the FPGA. JTAG is a relatively slow communication medium. You cannot rely on an FS2 console to stress test your memory interfaces. Refer to "Board Bring-up" on page 9–10 for strategies to stress test memory.

To learn more about the FS2 console refer to the First Silicon Solutions website at **www.fs2.com**. The Nios II Embedded Design Suite (EDS) installation also includes documentation for the FS2 console. You can find this documentation at: **$SOPC_KIT_NIOS2\bin\fs2\doc**.

System Console

You can use the System Console to perform low–level debugging of an SOPC Builder 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 SOPC Builder components with interfaces that the System Console can use to send commands and receive data. Table 9–1 lists these components.

Table 9–1. SOPC Builder Components for Communication with the System Console *(Note 1)*

Component Name	Debugs Components with the Following Interface Types
The 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.

Note to Table 9–1:

(1) The System Console can also send and receive byte streams from any SLD node, whether it is instantiated in an SOPC Builder component provided by Altera, a custom component, or part of your Quartus II project. However, this approach requires detailed knowledge of the JTAG commands.

The System Console allows you to perform any of the following tasks:

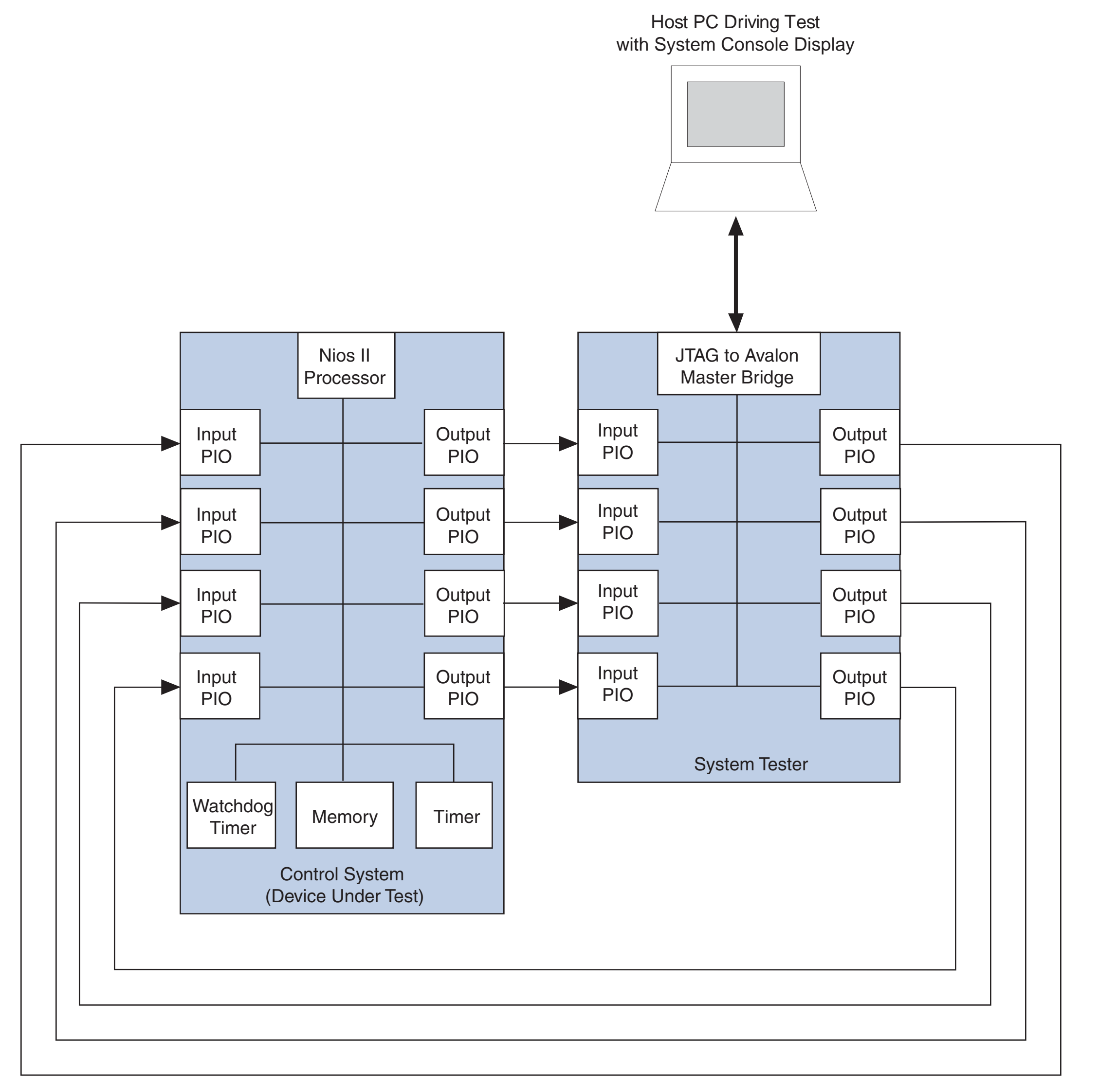
- Access memory and peripherals
- Start or stop a Nios II processor

- 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. Figure 9–3 illustrates HIL testing using the System Console.

Figure 9–3. Hardware-in-the-Loop Testing Using the System Console



To learn more about the System Console refer to the *System Console User Guide*.

SignalTap II Embedded Logic Analyzer

The SignalTap® II embedded logic analyzer is available in the Quartus II software. It reuses the JTAG pins of the FPGA and has a low Quartus II fitter priority, allowing it to be non-intrusive. Because this logic analyzer is integrated in your design automatically, it takes synchronized measurements without the undesirable side effects of output pin capacitance or I/O delay. The SignalTap II embedded logic analyzer also supports Tcl scripting so that you can automate data capture, duplicating the functionality that external logic analyzers provide.

This logic analyzer can operate while other JTAG components, including the Nios II JTAG debug module and JTAG UART, are in use, allowing you to perform co-verification. You can use the plug-in support available with the SignalTap II embedded logic analyzer to enhance your debug capability with any of the following:

- Instruction address triggering
- Non-processor related triggering
- Software disassembly
- Instruction display (in hexadecimal or symbolic format)

You can also use this logic analyzer to capture data from your embedded system for analysis by the MATLAB software from Mathworks. The MATLAB software receives the data using the JTAG connection and can perform post processing analysis. Using looping structures, you can perform multiple data capture cycles automatically in the MATLAB software, instead of manually controlling the logic analyzer using the Quartus II design software.

Because the SignalTap II embedded logic analyzer uses the FPGA's JTAG connection, continuous data triggering may result in lost samples. For example, if you capture data continuously at 100 MHz, you should not expect all of your samples to be displayed in the logic analyzer GUI. The logic analyzer buffers the data at 100 MHz; however, if the JTAG interface becomes saturated, samples are lost.

To learn more about SignalTap II embedded logic analyzer and co-verification, refer to the following documentation: *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook* and *AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems*.

External Instrumentation

If your design does not have enough on-chip memory to store trace buffers, you can use an external logic analyzer for debugging. External instrumentation is also necessary if you require any of the following:

- Data collection with pin loading
- Complex triggers
- Asynchronous data capture

Altera provides procedures to connect external verification devices such as oscilloscopes, logic analyzers, and protocol analyzers to your FPGA.

SignalProbe

The SignalProbe incremental routing feature allows you to route signals to output pins of the FPGA without affecting the existing fit of a design to a significant degree. You can use SignalProbe to investigate internal device signals without rewriting your HDL code to pass them up through multiple layers of the design hierarchy to a pin. Creating such revisions manually is time-consuming and error-prone.

Altera recommends SignalProbe when there are enough pins to route internal signals out of the FPGA for verification. If FPGA pins are not available, you have the following three alternatives:

- Reduce the number of pins used by the design to make more pins available to SignalProbe
- Use the SignalTap II embedded logic analyzer
- Use the Logic Analyzer Interface

Revising your design to increase the number of pins available for verification purposes requires design changes and can impact the design schedule. Using the SignalTap II embedded logic analyzer is a viable solution if you do not require continuous sampling at a high rate. The SignalTap II embedded logic analyzer does not require any additional pins to be routed; however, you must have enough unallocated logic and memory resources in your design to incorporate it. If neither of these approaches is viable, you can use the logic analyzer interface.

To learn more about SignalProbe, refer to the *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*.

Logic Analyzer Interface

The Quartus II Logic Analyzer Interface is a JTAG programmable method of driving multiple time-domain multiplexed signals to pins for external verification. Because the Logical Analyzer Interface multiplexes pins, it minimizes the pincount requirement. Groups of signals are assigned to a bank. Using JTAG as a communication channel, you can switch between banks.

You should use this approach when SignalTap II embedded logic analyzer is insufficient for your verification needs. Some external logic analyzer manufacturers support the Logic Analyzer Interface. These logic analyzers have various amounts of support. The most important feature is the ability to let the measurement tools cycle through the signal banks automatically.

The ability to cycle through signal banks is not limited to logic analyzers. You can use it for any external measurement tool. Some developers use low speed indicators, for example LEDs, for verification. You can use the Logic Analyzer interface to map many banks of signals to a small number of verification LEDs. You may wish to leave this form of verification in your final design so that your product is capable of creating low-level error codes after deployment.

To learn more about the Quartus II Logic Analyzer Interface, refer to the *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*.

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 document. The following recommendations on creating effective seed values should help you avoid repeating data values:

- Use a random noise measurement. One way to do this is by reading the analog output value of an A/D converter.
- Use multiple asynchronous counters in combination to create seed values.
- Use a timer value as the seed (that is, the number of seconds from a fixed point in time).

Using a combination of seed generation techniques can lead to more random behavior. When generating random sequences, it is important to understand the distribution of the random data generated. Some generators create linear sequences in which the distribution is evenly spread across the random number domain. Others create non-linear sequences that may not provide the test coverage you require. Before you begin using a random number generator to verify your system, examine the data created for a few sequences. Doing so helps you understand the patterns created and avoid using an inappropriate set of inputs.

Board Bring-up

You can minimize board bring-up time by adopting a systematic strategy. First, break the task down into manageable pieces. Verify the design in segments, not as a whole, beginning with peripheral testing.

Peripheral Testing

The first step in the board bring-up process is peripheral testing. Add one interface at a time to your design. After a peripheral passes the tests you have created for it, you should remove it from the test design. Designers typically leave the peripherals that pass testing in their design as they move on to test other peripherals. Sometimes this is necessary; however, it should be avoided when possible because multiple peripherals can create instability due to noise or crosstalk. By testing peripherals in a system individually, you can isolate the issues in your design to a particular interface.

A common failure in any system is involves memory. The most problematic memory devices operate at high speeds, which can result in timing failures. High performance memory also requires many board traces to transfer data, address, and control signals, which cause failures if not routed properly. You can use the Nios II processor to verify your memory devices using verification software or a debugger such as the FS2 console. 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 the *Embedded Design Handbook*.

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. Table 9–2 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 9–2. ***Walking Ones Example***

Written Value	Read Value	Failure Detected
00000001	00000001	No failure detected
00000010	00000000	Error, most likely the second data bit, D[1] stuck low or shorted to ground
00000100	00000100	No failure detected, confirmed D[1] is stuck low or shorted to another trace that is not listed in this table.
00001000	00001000	No failure detected
00010000	00010000	No failure detected
00100000	01100000	Error, most likely D[6] and D[5] short circuited
01000000	01100000	Error, confirmed that D[6] and D[5] are short circuited
10000000	10000000	No failure detected

Address Trace Failure

The address trace test is similar to the walking ones test used for data with one exception. For this test you must write to all the test locations before reading back the data. Using address locations that are powers of two, you can quickly verify all the address traces of your circuit board.

The address trace test detects the aliasing effects that short or open circuits can have on your memory interface. For this reason it is important to write to each location with a different data value so that you can detect the address aliasing. You can use increasing numbers such as 1, 2, 3, 4, and so on while you verify the address traces in your system. Table 9–3 shows how to use powers of two in the process of finding an address trace failure:

Table 9–3. Powers of Two Example

Address	Written Value	Read Value	Failure Detected
00000000	1	1	No failure detected
00000001	2	2	No failure detected
00000010	3	1	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	6	Error, A[5] and A[4] are short circuited
01000000	8	8	No failure detected
10000000	9	9	No failure detected

Device Isolation

Using device isolation techniques, you can disable features of devices on your PCB that cause your design to fail. Typically designers use device isolation for early revisions of the PCB, and then remove these capabilities before shipping the product.

Most designs use crystal oscillators or other discrete components to create clock signals for the digital logic. If the clock signal is distorted by noise or jitter, failures may occur. To guard against distorted clocks, you can route alternative clock pins to your FPGA. If you include SMA connectors on your board, you can use an external clock generator to create a clean clock signal. Having an alternative clock source is very useful when debugging clock-related issues.

Sometimes the noise generated by a particular device on your board can cause problems with other devices or interfaces. Having the ability to reduce the noise levels of selected components can help you determine the device that is causing issues in your design. The simplest way to isolate a noisy component is to remove the power source for the device in question. For devices that have a limited number of power pins, if you include 0 ohm resistors in the path between the power source and the pin. You can cut off power to the device by removing the resistor. This strategy is typically not possible with larger devices that contain multiple power source pins connecting directly to a board power plane.

Instead of removing the power source from a noisy device, you can often put the device into a reset state by driving the reset pin to an active state. Another option is to simply not exercise the device so that it remains idle.

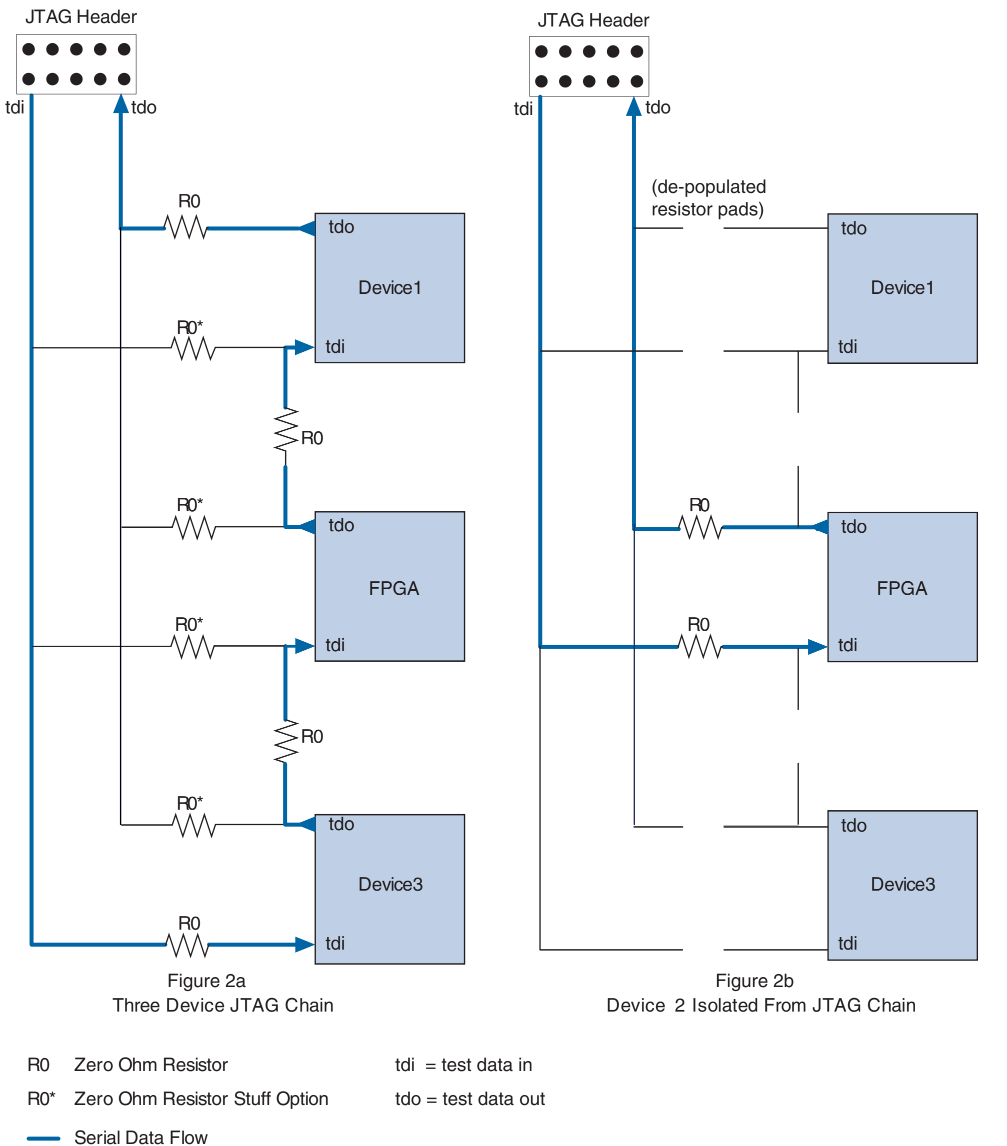
A noisy power supply or ground plane can create signal integrity issues. With the typical voltage swing of digital devices frequently below a single volt, the power supply noise margin of devices on the PCB can be as little as 0.2 volts. Power supply noise can cause digital logic to fail. For this reason it is important to be able to isolate the power supplies on your board. You can isolate your power supply by using fuses that are removed so that a stable external power supply can be substituted temporarily in your design.

JTAG

FPGAs use the JTAG interface for programming, communication, and verification. Designers frequently connect several components, including FPGAs, discrete processors, and memory devices, communicating with them through a single JTAG chain. Sometimes the JTAG signal is distorted by electrical noise, causing a communication failure for the entire group of devices. To guarantee a stable connection, you must isolate the FPGA under test from the other devices in the same JTAG chain.

Figure 9–4a 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 Figure 9–4b illustrates. This technique allows you to isolate one device while using a single JTAG chain.

Figure 9–4. JTAG Isolation



To learn more about JTAG refer to the *IEEE 1149.1(JTAG) Boundary-Scan Testing in Altera Devices*.

Board Testing

You should convert the simulations you run to verify your intellectual property (IP) before fabrication to test vectors that you can then run on the hardware to verify that the simulation and hardware versions exhibit the same behavior. Manufacturing can also use these tests as part of a regularly scheduled quality assurance test. Because the tests are run by engineers in other organizations they must be documented and easy to run.

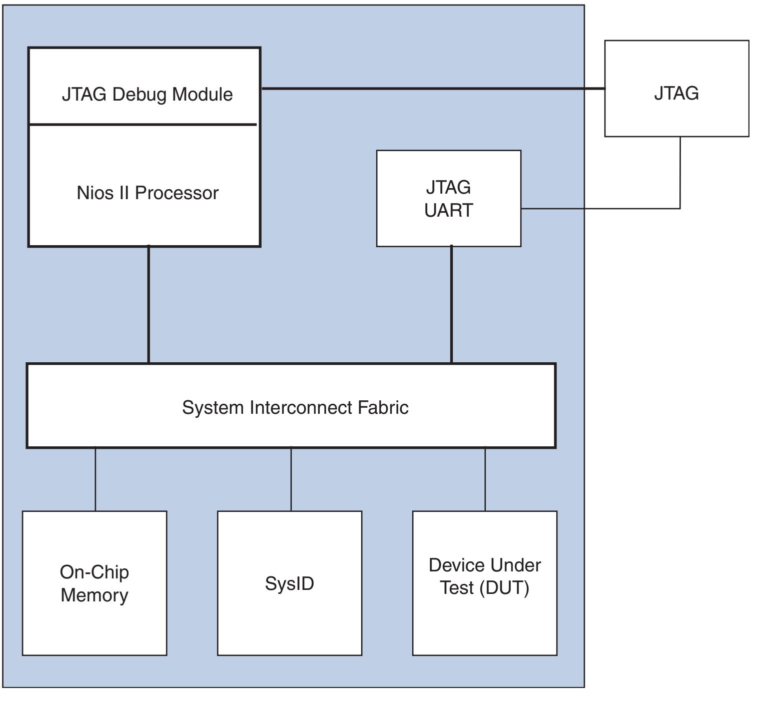
Minimal Test System

Whether you are creating your first embedded system in a FPGA, or are debugging a complex issue, you should always begin with a minimal system. To minimize the probability of signal integrity issues, reduce the pincount of your system to the absolute minimal number of required pins. In an embedded design that includes the Nios II processor, the minimal pincount might be clock and reset signals. Such a system might include the following the following components:

- Nios II processor (with a level 1 debug core)
- On-chip memory
- JTAG UART
- System ID core

Using these four components you can create a functioning embedded system including debug and terminal access. To simplify your debug process, you should use a Nios II processor that does not contain a data cache. The Nios II/e and Nios II/s cores do not include data caches. The Nios II/f core can also be configured without a data cache. Figure 9–5 illustrates a minimal system. In this system, you have to route only the clock pin and reset pins, because the JTAG signals are automatically connected by the Quartus II software.

Figure 9–5. Simple Test System



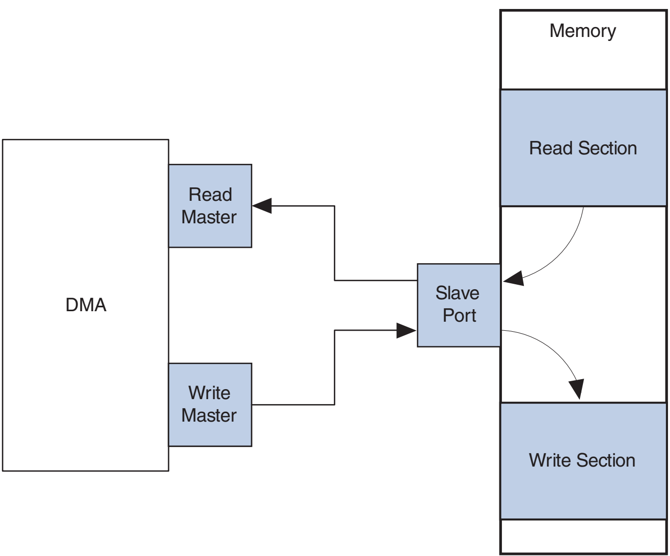
You can use the Nios II JTAG debug module to download software to the processor. Before testing any additional interfaces you should execute a small program that prints a message to the terminal to verify that your minimal system is functioning properly.

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
- A Nios II JTAG debug module and FS2 console
- The SignalTap II embedded logic analyzer
- An external logic interface
- SignalProbe
- A direct memory access (DMA) engine
- In-system updating of memory and constants

The Nios II processor is not capable of stress testing high speed memory devices. Altera recommends that you use a DMA engine to stress test memories. A stress test should access memory as frequently as possible, performing continuous reads or writes. Typically, the most problematic access sequence for high-speed memory involves the bus turnaround between read and write accesses. You can test these cases by connecting the DMA read and write masters to the same memory and transferring the contents from one location to another, as shown in Figure 9–6.

Figure 9–6. 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 or use the Nios II C2H Compiler to create hardware used for testing.

To learn more about the topics covered in this section refer to the following documentation:

- Nios II Hardware Development Tutorial
- Quartus II Verification Methods web page
- *DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *In-System Updating of Memory and Constants* chapter in volume 3 of the *Quartus II Handbook*

System Verification

System verification is the final step of system design. This section focuses on common mistakes designers make during system verification and methods for correcting and avoiding them. It includes the following topics:

- Designing with Verification in Mind
- Accelerating Verification
- Using Software to Verify Hardware
- Environmental Testing

Designing with Verification in Mind

As you design, you should focus on both the development tasks and the verification strategy. Doing so results in a design that is easier to verify. If you create large, complicated blocks of logic and wait until the HDL code is complete before testing, you spend more time verifying your design than if you verify it one section at a time.

Consider leaving in verification code after the individual sections of your design are working. If you remove too much verification logic it becomes very difficult to reintroduce it at a later time if needed. If you discover an issue during system integration, you may need to revisit some of the smaller block designs. If you modify one of the smaller blocks, you must re-test it to verify that you have not created additional issues.

Designing with verification in mind is not limited to leaving verification hooks in your design. Reserving enough hardware resources to perform proper verification is also important. The following recommendations can help you avoid running out of hardware resources:

- Design and verify using a larger pin-compatible FPGA.
- Reserve hardware resources for verification in the design plan.
- Design the logic so that optional features can be removed to free up verification resources.

Finally, schedule a nightly regression test of your design to increase your test coverage between hardware or software compilations.

Accelerating Verification

Altera recommends the verification flow illustrated in Figure 9–7. 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.

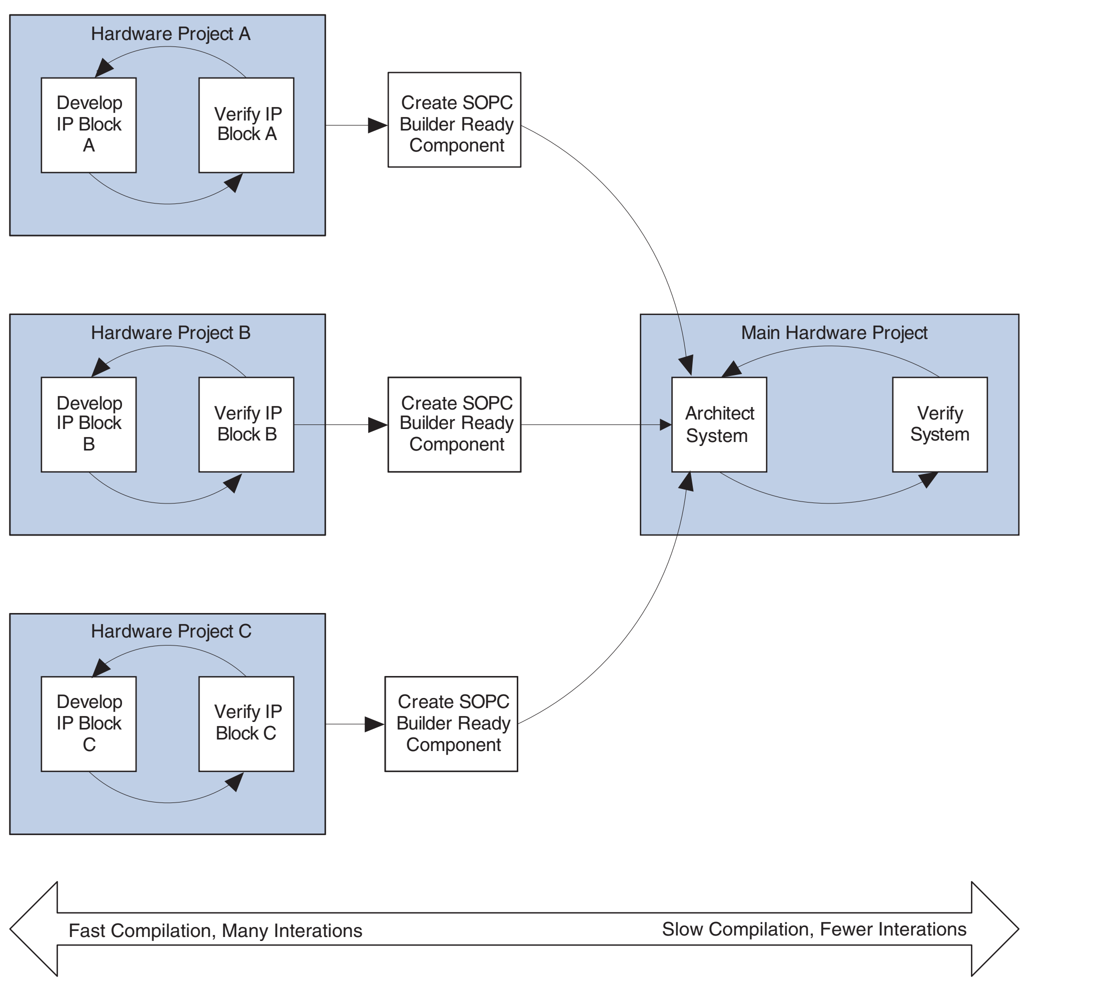
After the individual components are verified, you can integrate them in an SOPC Builder system. The integrated system must include an Avalon-MM or Avalon Streaming (Avalon-ST) port. Using the component editor available from SOPC Builder, you add an Avalon-MM interface to your existing component and integrate it in your system.

After your system is created in SOPC Builder, 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 9–7. IP Verification and Integration Flow



To learn more about component editor and system integration, refer to the following documentation:

- The *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- The *SOPC Builder Component Development Walkthrough* chapter in volume 4 of the *Quartus II Handbook*
- The *Avalon Interface Specifications*

Using Software to Verify Hardware

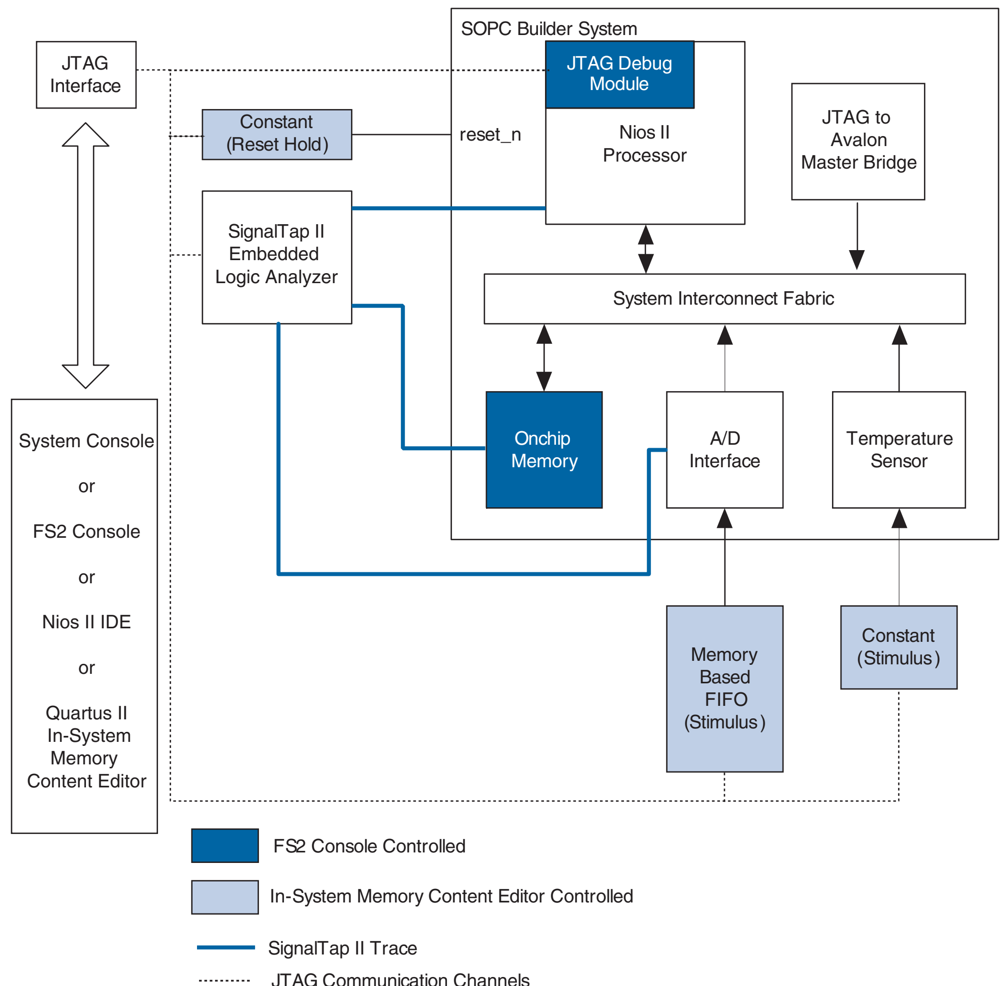
Many hardware developers use test benches and test harnesses to verify their logic in simulations. These strategies can be very time consuming. Instead of relying on simulations for all your verification tasks, you can test your logic using software or scripts, as Figure 9–8 illustrates.

This system uses the JTAG interface to access components connected to the system interconnect fabric and to create stimuli for the system. If you use the JTAG server provided by the Quartus II 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 IDE. You can also use the Nios II JTAG debug core to transmit files to and from your embedded system using the host file system. Using the System Console you can access components in your system and also run scripts for automated testing purposes.

Using the Quartus II 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. This approach is similar to using the FS2 console to control logic in your system. However, unlike the FS2 console, you can use the In-System Memory Content Editor to access hardware that is not memory-mapped. All of the verification techniques described in this chapter 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. *Developing Software for Nios II* in the *Embedded Design Handbook* also includes a significant amount of information about the system file system.

Figure 9–8. Script Controlled Verification



To learn more about the verification and scripting abilities outlined in the example above, refer to the following documentation:

- First Silicon Solutions Website, **www.fs2.com**
- Altera Basic Quartus II Tcl Scripting training course
- *Quartus II Scripting Reference Manual*

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.

Referenced Documents

This chapter references the following documents:

- *AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems*
- *Avalon Interface Specifications*
- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *Debugging Nios II Designs* chapter in the *Embedded Design Handbook*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* in volume 3 of the *Quartus II Handbook*
- *Developing Nios II Software* in the *Embedded Design Handbook*
- *IEEE 1149.1(JTAG) Boundary-Scan Testing in Altera Devices*
- *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*
- *In-System Updating of Memory and Constants* in volume 3 of the *Quartus II Handbook*
- *Nios II Hardware Development Tutorial*
- *Quartus II Scripting Reference Manual*
- *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*
- *SOPC Builder Component Development Walkthrough* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 9–4 shows the revision history for this chapter.

Table 9–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2008 v1.2	■ In the FS2 Console section, added `sld info` command and an example that writes and reads a range of memory addresses. ■ Added introductory discussion to the System Console. ■ Added JTAG to Avalon Master Bridge to Figure 9–8.	Updated to provide more information about the FS2 Console and introduce the System Console.
June 2008 v1.1	Corrected Table of Contents.	—
March 2008 v1.0	Initial release.	—

10. Interfacing an External Processor to an Altera FPGA

ED51011-1.0

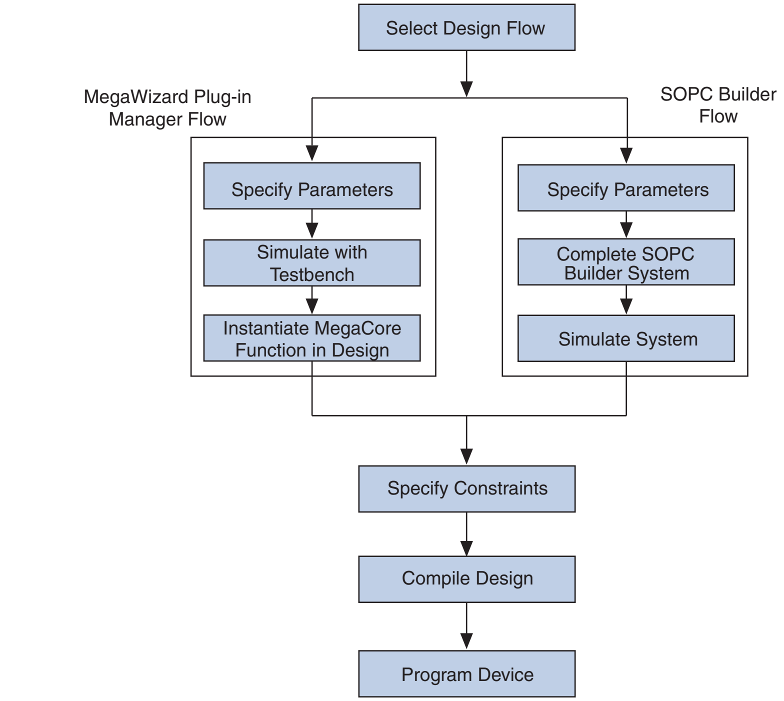
This chapter provides an overview of the options Altera® provides to connect an external processor to an Altera FPGA or Hardcopy® device. These interface options include the PCI Express, PCI, RapidIO®, serial peripheral interface (SPI) interface or a simple custom bridge that you can design yourself.

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

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

You can instantiate the PCI Express, PCI, and RapidIO MegaCore functions using either the MegaWizard™ Plug-In Manager or SOPC Builder design flow. The PCI Lite and SPI cores are only available in the SOPC Builder design flow. SOPC Builder automatically generates the HDL design files that include all of the specified components and system connectivity. Alternatively, you can use the MegaWizard Plug-In Manager to generate a stand-alone component outside of SOPC Builder. Figure 10–1 shows the steps you take to instantiate a component in both design flows.

Figure 10–1. SOPC Builder and MegaWizard Plug-In Manager Design Flows



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

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

Configuration Options

Figure 10–2 illustrates an SOPC Builder system design that includes a high-performance external bus or switch to connect an industry-standard processor to an external interface of a MegaCore function inside the FPGA. This MegaCore function also includes an Avalon-MM master port that connects to the SOPC Builder system interconnect fabric. As Figure 10–2 illustrates, Altera provides a library of components, typically Avalon-MM slave devices, that connect seamlessly to the Avalon system interconnect fabric.

Figure 10–2. ***FPGA with a Bus or Switch Interface Bridge for Peripheral Expansion***

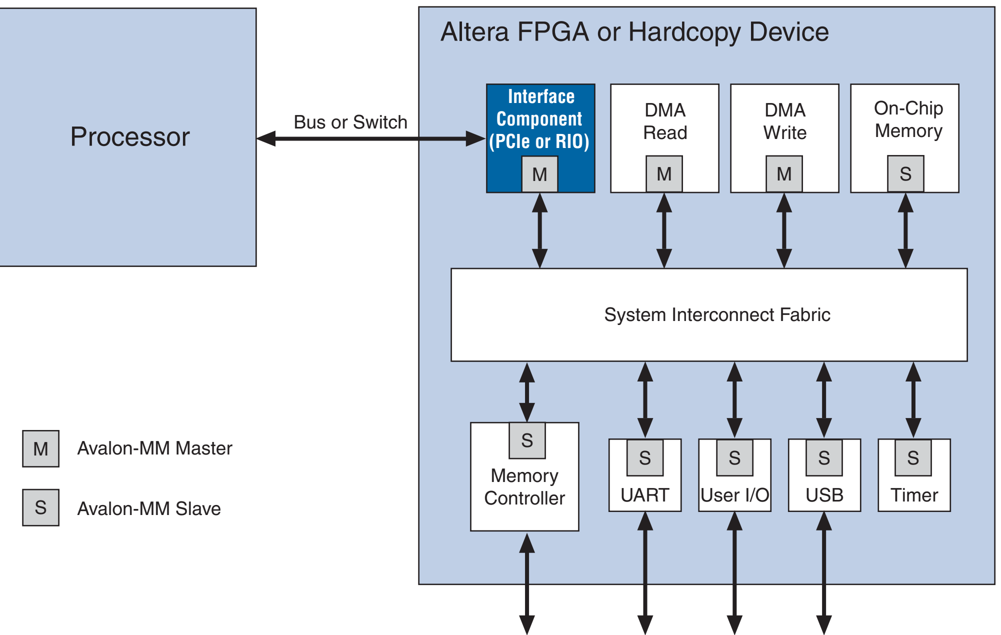
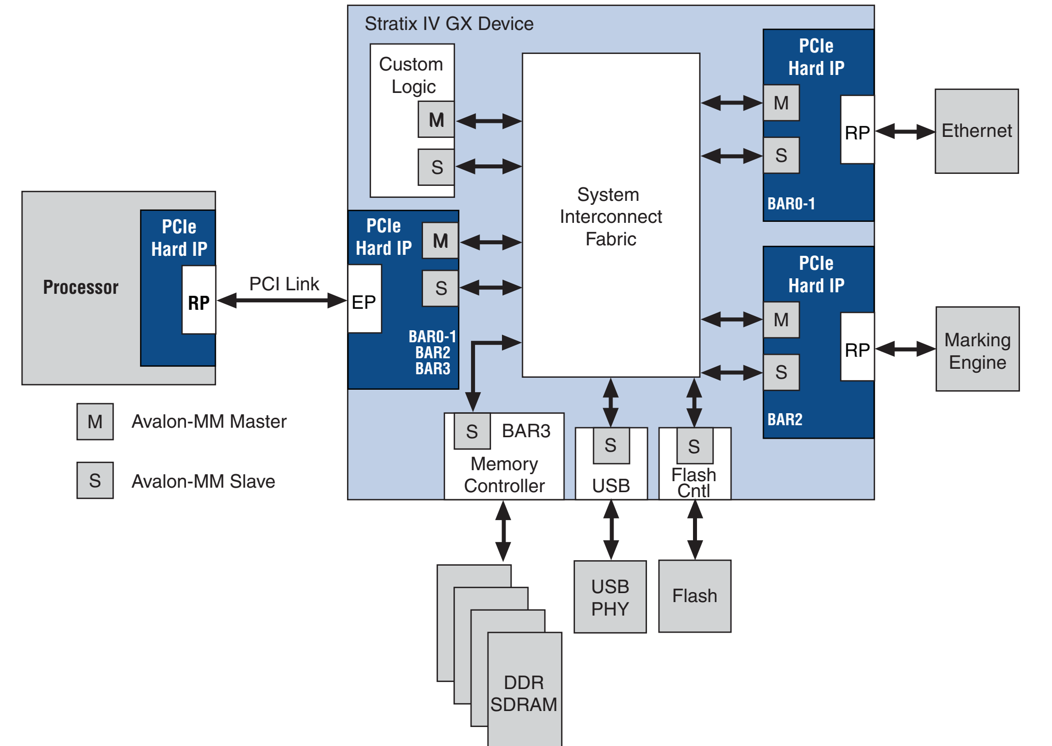


Figure 10–3 illustrates a design that includes an external processor that interfaces to an PCI Express endpoint inside the FPGA. The system interconnect fabric inside the implements a partial crossbar switch between the endpoint that connects to the external processor and two additional PCI Express root 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. SOPC Builder automatically generates the system interconnect fabric to connect the components in the system.

Figure 10–3. FPGA with a Processor Bus or SPI for Peripheral Expansion



Alternatively, you can also implement your logic in Verilog HDL or VHDL without using SOPC Builder. Figure 10–4 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 10–4. FPGA Performs Co-Processing

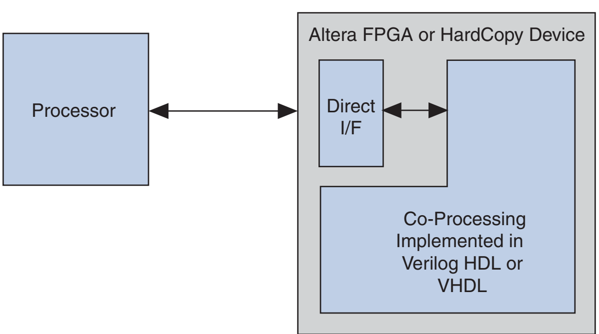


Table 10–1 summarizes the components Altera provides to connect an Altera FPGA or HardCopy device to an external processor. As this table indicates, three of the components are also available for use in the MegaWizard Plug-In Manager design flow in addition to the SOPC Builder. Alternative implementations of these components are also available through the Altera Megafunction Partners Program (AMPP℠) partners. The AMPP partners offer a broad portfolio of megafunctions optimized for Altera devices.

For a complete list of third-party IP for Altera FPGAs, refer to the IP MegaStore web page: **www.altera.com/products/ip/ipm-index.html**. For SOPC Builder components, search for `sopc_builder_ready` in the IP MegaStore megafunction search function

Table 10–1. Processor Interface Solutions Available from an Altera

Protocol	Available in SOPC Builder	Available In MegaWizard Plug-In Manager	Third-Party Solution	OpenCore Plus Evaluation Available
RapidIO	✓	✓	✓	✓
PCI Express	✓	✓	✓	✓
PCI	✓	✓	✓	✓
PCI Lite	✓	—	—	License not required.
SPI	✓	—	—	

Table 10–2 summarizes the most popular options for peripheral expansion in SOPC Builder systems that include an industry-standard processor. All of these are available in SOPC Builder. Some are also available using the MegaWizard Plug-In Manager.

Table 10–2. Partial list of peripheral interfaces available for SOPC Builder

Protocol	Available in SOPC Builder	Available In MegaWizard Plug-In Manager	Third-Party Solution	OpenCore Plus Evaluation Available
CAN	✓	—	✓	✓
I2C	✓	—	✓	✓
Ethernet	✓	✓	✓	✓
PIO	✓	—	—	Not required
POS-PHY Level 4 (SPI 4.2)	—	✓	—	✓
SPI	✓	—	✓	Not required
UART	✓	—	✓	✓
USB	✓	—	✓	✓

For detailed information on the components available in SOPC builder refer to *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

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

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

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

RapidIO Interface

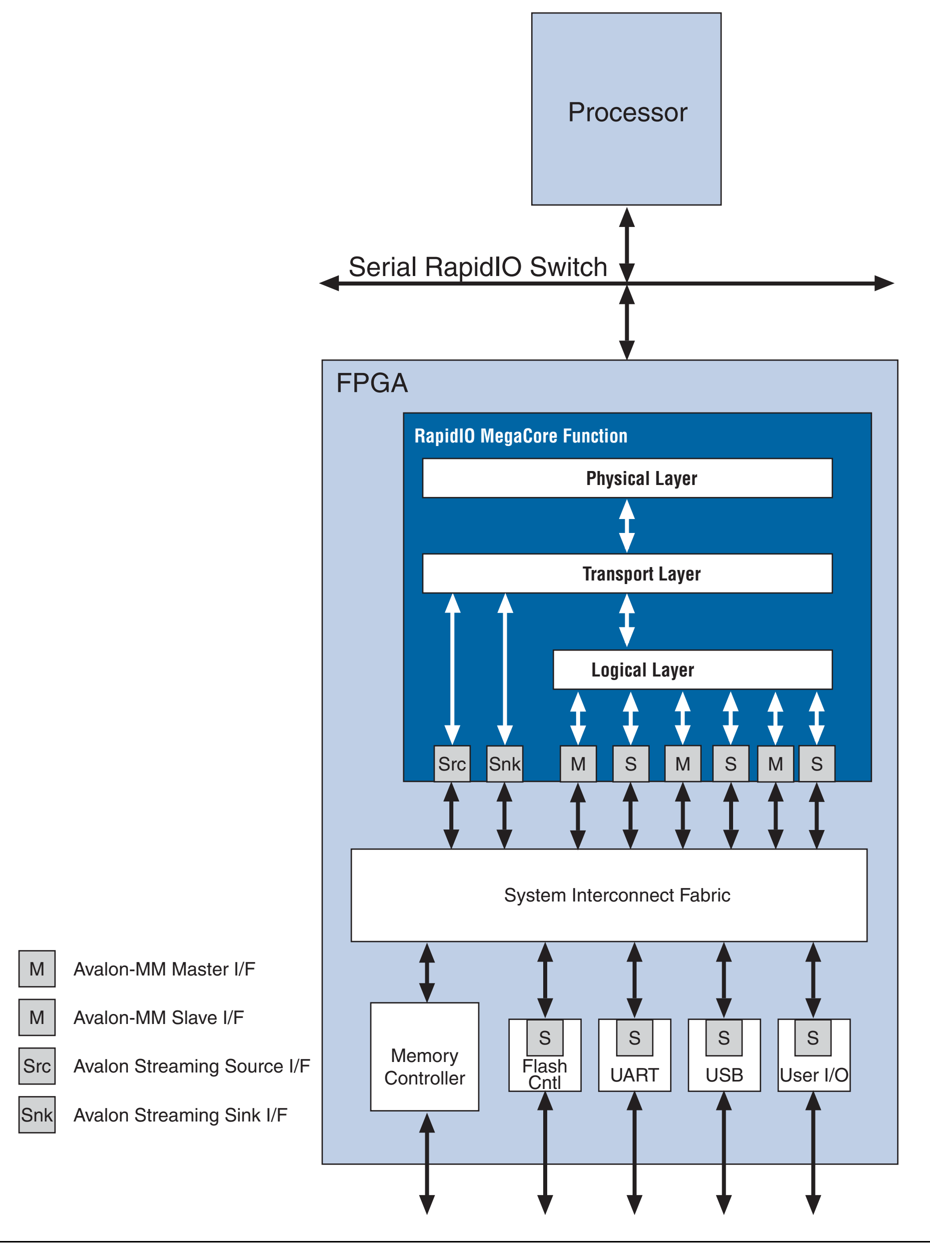
RapidIO is a high-performance packet-switched protocol that transports data and control information between processors, memories, and peripheral devices. The RapidIO MegaCore function is available in SOPC Builder includes Avalon-MM ports that translate Serial RapidIO transactions into Avalon-MM transactions.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

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

- Avalon Streaming sink pass-through Tx
- Avalon-ST source pass-through Rx

Using the SOPC Builder design flow, you can integrate a RapidIO endpoint into an SOPC Builder system. You connect the ports using the SOPC Builder **System Contents** tab and SOPC Builder automatically generates the system interconnect fabric. Figure 10–5 illustrates an SOPC Builder system that includes a processor and a RapidIO MegaCore function.

Figure 10–5. Example system with RapidIO Interface



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

Refer to the following documents for a complete description of the RapidIO MegaCore function:

- *RapidIO MegaCore Function User Guide*
- *AN 513: RapidIO Interoperability With TI 6482 DSP Reference Design*

PCI Express Interface

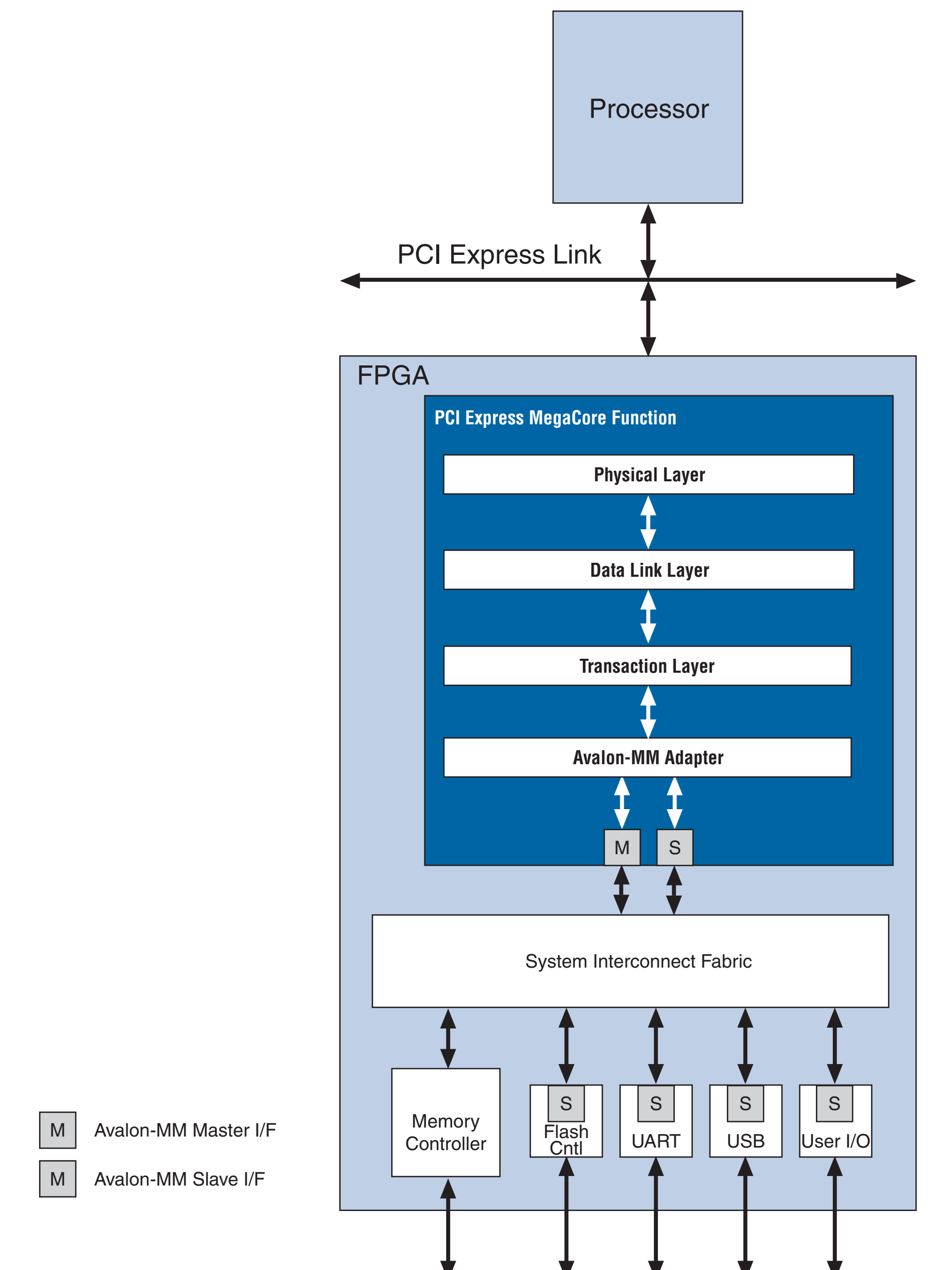
The PCI Express MegaCore function configured using the SOPC Builder design flow uses the PCI Express Compiler's Avalon-MM bridge module to connect the PCI Express component to the system interconnect fabric. The bridge facilitates the design of PCI Express systems that use the Avalon-MM interface to access SOPC Builder components. Figure 10–6 illustrates a design that links an external processor to an SOPC Builder system using the PCI Express MegaCore function.

You can also implement the PCI Express MegaCore function using the MegaWizard Plug-In Manager design flow. The configuration options for the two design flows are different. The PCI Express MegaCore function is available in Stratix IV and Arria II GX devices as a hard IP implementation and can be used as a root port or end point.

For more information about using the PCI Express MegaCore function refer to the following documents:

- *PCI Express Compiler User Guide*
- *AN 513: SOPC Builder PCI Express Design with GUI Interface*
- *AN 456: PCI Express High Performance Reference Design*
- *AN 443: External PHY Support in PCI Express MegaCore Functions*
- *AN 431: PCI Express-to-DDR2 SDRAM Reference Design.*

Figure 10–6. Example system with PCI Express interface

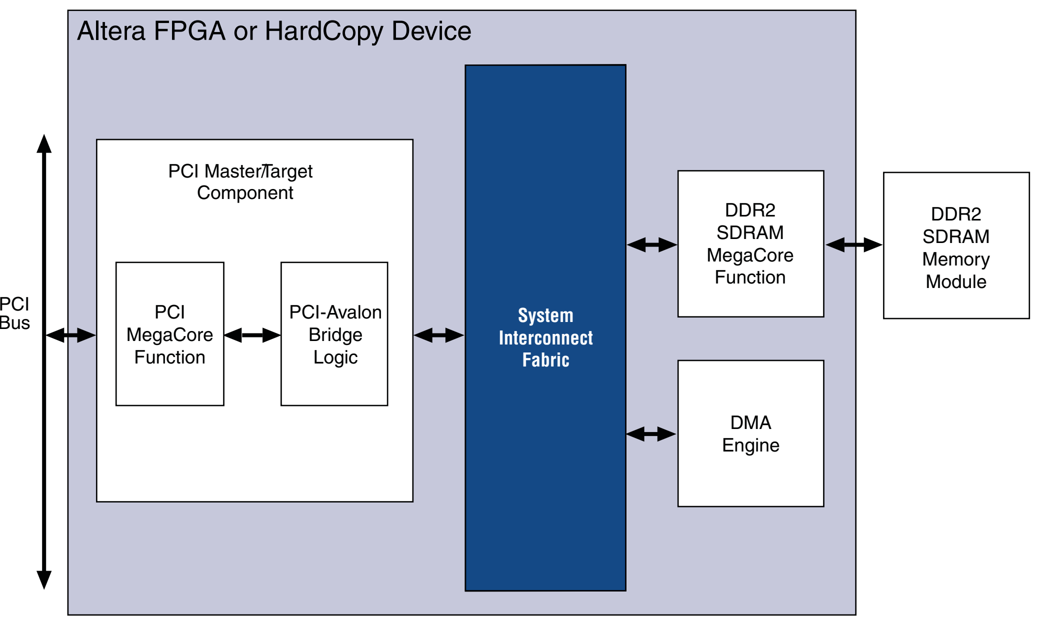


PCI Interface

Altera offers a wide range of PCI local bus solutions that you can use to connect a host processor to an FPGA. You can implement the PCI MegaCore function using the MegaWizard Plug-In Manager or SOPC Builder design flow.

The PCI SOPC Builder 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. Figure 10–7 illustrates an SOPC Builder system using the PCI MegaCore function. You can parameterize the PCI MegaCore function with a 32- or 64-bit interface.

Figure 10–7. PCI MegaCore Function in an SOPC Builder System



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

PCI Lite Interface

The PCI Lite component is optimized for low-latency and high throughput designs. It is available only in the SOPC Builder design flow. The PCI Lite core provides a subset of the PCI MegaCore function feature set to obtain a low-latency path that interfaces to a processor and other peripherals connected to the system interconnect fabric in an FPGA. This component translates PCI transactions to Avalon-MM transactions. The PCI Lite core uses the PCI-Avalon bridge to connect the PCI bus to the system interconnect fabric, allowing you to easily create simple PCI systems that include one or more SOPC Builder components.

For more information refer to the *PCI Lite Core* chapter in volume 5 of the *Quartus II Handbook*.

You can also implement the original PCI master/target and target MegaCore functions without an Avalon-MM bridge module using the MegaWizard Plug-In Manager design flow.

For information, refer to following reference designs:

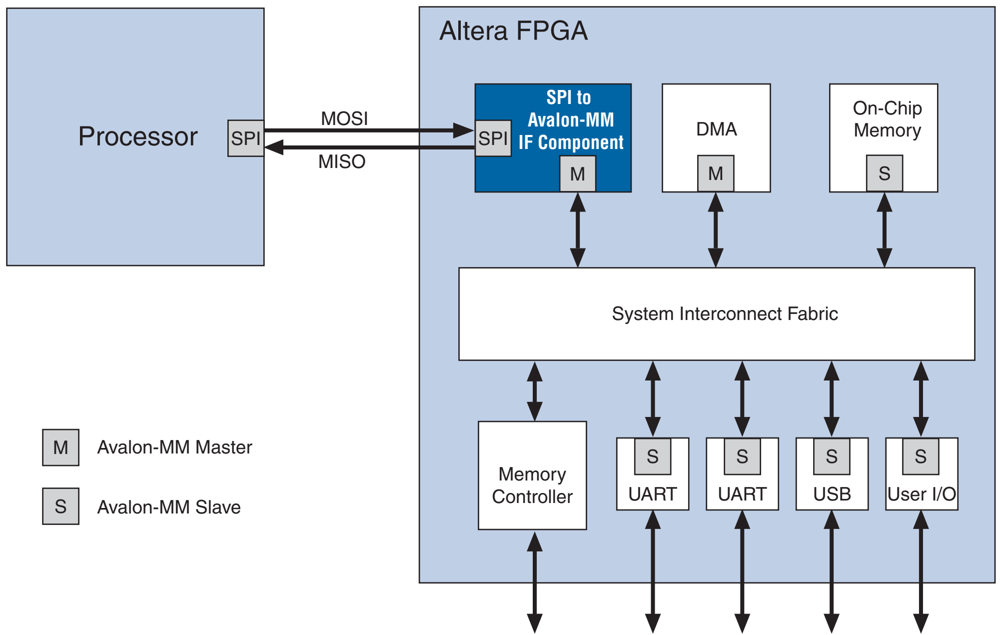
- *AN 390: PCI-to-DDR2 SDRAM Reference Design*
- *AN 223: PCI-to-DDR SDRAM Reference Design*

Serial Protocol Interface (SPI)

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

The SPI Slave to Avalon Master Bridge is an SOPC Builder-ready component that integrates easily into any SOPC Builder 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 in volume 5 of the *Quartus II Handbook*.

Figure 10–8. Example System with SPI to Avalon-MM Interface Component



Details of each protocol layer can be found the following documentation:

SPI Slave/JTAG to Avalon Master Bridge Cores—Provide a connection from an external host system to an SOPC Builder 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 SOPC Builder 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.

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 SOPC Builder 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 SOPC Builder to quickly define a custom bridge component to adapt the external processor bus to the Avalon-MM interface or any of the 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 *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

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.

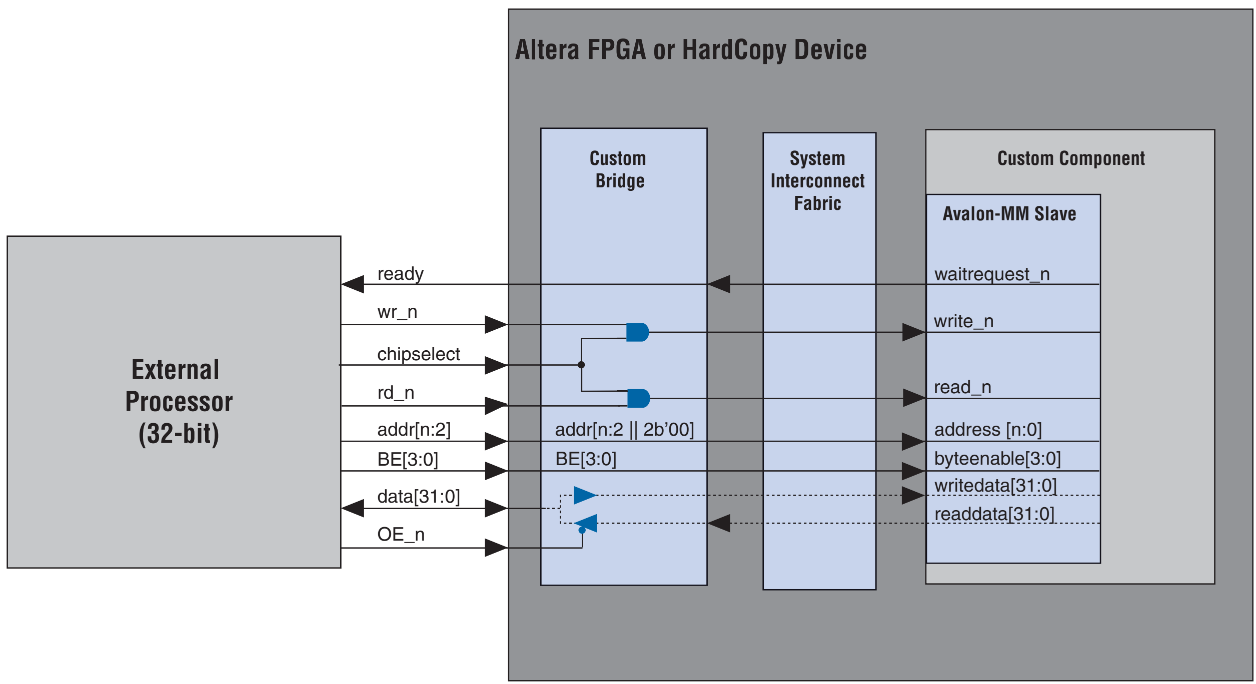
Consider the following points when designing a custom bridge to interface between an external processor and the Avalon-MM interface:

- 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 SOPC Builder 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. Figure 10–9 illustrates the use of a bridge component with a 32-bit external processor.

Figure 10–9. Custom Bridge to Adapt an External Processor to an Avalon-MM Slave Interface



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

Conclusion

Altera offers a variety of components that you can use to connect an FPGA to an external processor. With most of these components, you can choose either the SOPC Builder or MegaWizard Plug-In Manager design flow. You can also build your own custom interface to an external processor. By using the Avalon-MM interface in SOPC Builder, you can easily extend system capabilities for processors by taking advantage of the SOPC Builder library of components.

Referenced Documents

This chapter references the following documents:

- *AN 223: PCI-to-DDR SDRAM Reference Design*
- *AN320 - OpenCore Plus Evaluation of Megafunctions*
- *AN343 - OpenCore Evaluation of AMPP Megafunctions*
- *AN 390: PCI-to-DDR2 SDRAM Reference Design*
- *AN 431: PCI Express-to-DDR2 SDRAM Reference Design*
- *AN 443: External PHY Support in PCI Express MegaCore Functions*
- *AN 456: PCI Express High Performance Reference Design*
- *AN 513: RapidIO Interoperability With TI 6482 DSP Reference Design*
- *AN532: An SOPC Builder PCI Express Design with GUI Interface*
- *Avalon Interface Specifications*
- *Avalon Packets to Transactions Converter Core*
- *Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores*
- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *PCI Compiler User Guide*
- *PCI Express Compiler User Guide*
- *PCI Lite Core* chapter in volume 5 of the *Quartus II Handbook*
- *RapidIO MegaCore Function User Guide*
- *SPI Slave/JTAG to Avalon Master Bridge Core*
- *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*

Document Revision History

Table 10–3 shows the revision history for this chapter.

Table 10–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
February 2009, v1.0	Initial release	—

Additional Information

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact *(Note 1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>*.**pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
“Subheading Title”	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: “Typographic Conventions.”
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.

Visual Cue	Meaning
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information about a particular topic.