Creating Multiprocessor Nios II Systems

Tutorial

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

TU-N2033005-1.4

Document Version: 1.4
Document Date: February 2010

Copyright © 2010 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.

NSAI

I.S. EN ISO 9001

Contents

Chapter 1. Creating Multiprocessor Nios II Systems

Introduction 1-1
Benefits of Multiprocessor Systems 1-2
Nios II Multiprocessor Systems 1-2
Hardware Design Considerations 1-3
Autonomous Multiprocessors 1-3
Multiprocessors that Share Resources 1-4
Sharing Resources in a Multiprocessor System 1-4
Sharing Memory 1-6
The Hardware Mutex and Mailbox Cores 1-7
Sharing Peripherals Between Multiple Processors 1-8
Multiprocessors and Overlapping Address Space 1-9
Software Design Considerations 1-10
Program Memory 1-10
Boot Addresses 1-12
Running and Debugging Multiprocessor Systems from the Nios II SBT for Eclipse 1-14
Design Example 1-15
Hardware and Software Requirements 1-15
Installation Notes 1-15
Creating the Hardware System 1-15
Creating Software for the Multiprocessor System 1-24
Building the Application and BSP Projects 1-24
Starting the Nios II SBT for Eclipse 1-25
Importing the Software Projects 1-25
Building the Software Projects 1-26
Creating a Debug Configuration for Each Processor 1-26
Debugging the Software Projects on the Board 1-27
Conclusion 1-27

Additional Information

Revision History Info-1
Referenced Documents Info-1
How to Contact Altera Info-1
Typographic Conventions Info-2

1. Creating Multiprocessor Nios II Systems

This tutorial describes the features of the Altera® Nios® II processor and SOPC Builder tool that are useful for creating systems with two or more processors. The tutorial provides an example design that guides you through a step-by-step process for building a multiprocessor system containing two processors that access a shared memory buffer using a mailbox. It shows you how to use the Nios II Software Build Tools (SBT) for Eclipse to create and debug two software projects, one for each processor in the system.

Refer to the *Nios II Embedded Design Suite Release Notes and Errata* and the *MegaCore IP Library Release Notes and Errata* for the latest features, enhancements, and known issues in the current release.

Introduction

Any system that incorporates two or more microprocessors working together to perform one or more related tasks is commonly referred to as a multiprocessor system. Developers using the Altera Nios II processor and SOPC Builder tool can quickly design and build multiprocessor systems that share resources. SOPC Builder is a system development tool for creating SOPC design systems that can include processors, peripherals, and memories. A Nios II processor system typically refers to a system with a processor core, a set of on-chip peripherals, on-chip memory and interfaces to off-chip memory all implemented on a single Altera device.

This document describes the features of the Nios II processor and SOPC Builder tool that are useful for creating systems with two or more processors. This document provides an example design that guides you through a step-by-step process for building a multiprocessor system containing two processors that share a mail box and a memory buffer. Using the Nios II SBT, you create two software projects, one for each processor in the system. Then you import these two software projects to the Nios II SBT for Eclipse, and use the Nios II SBT for Eclipse to debug these two software projects.

After completing this document, you will have the knowledge to perform the following tasks:

- Build an SOPC Builder system containing more than one Nios II processor.
- Safely share resources between processors, avoiding data corruption.
- Build software projects for multiprocessor systems using the Nios II SBT.
- Debug multiple software projects running on multiple processors using the Nios II SBT for Eclipse.

This chapter assumes that you are familiar with reading and writing embedded software and that you have read and followed the step-by-step procedures for building a microprocessor system in the *Nios II Hardware Development Tutorial*.

The *Nios II Hardware Development Tutorial* can be found on the Literature: Nios II Processor page.

Benefits of Multiprocessor Systems

Multiprocessor systems possess the benefit of increased performance, but nearly always at the price of significantly increased system complexity. For this reason, the use of multiprocessor systems has historically been limited to workstation and high-end PC computing using a complex method of load-sharing often referred to as symmetric multiprocessing (SMP). While the overhead of SMP is typically too high for most embedded systems, the idea of using multiple processors to perform different tasks and functions on different processors in embedded applications (asymmetrical) is gaining popularity. Altera FPGAs provide an ideal platform for developing asymmetric embedded multiprocessor systems, because the hardware can easily be modified and tuned using the SOPC Builder tool to provide optimal system performance. Recent increases in the size of Altera FPGAs make possible system designs with many Nios II processors on a single chip. Furthermore, with a powerful integration tool like SOPC Builder, different system configurations can be designed, built, and evaluated very quickly.

Nios II Multiprocessor Systems

The Nios II SBT for Eclipse includes features to help with the creation and debugging of multiprocessor systems. Multiple Nios II processors are able to efficiently share system resources thanks to the multimaster friendly slave-side arbitration capabilities of the system interconnect fabric. Since the capabilities of SOPC Builder now allow users to almost effortlessly add as many processors to a system as desired, the design focus in building multiprocessor systems no longer lies in the arranging and connecting of hardware components. The challenge in building multiprocessor systems now lies in writing the software for those processors so they operate efficiently together, and do not conflict with one another.

To aid in the prevention of multiple processors interfering with each other, multiprocessor coordination peripherals, such as a hardware mutex core and a hardware mailbox core, are included in the Nios II Embedded Design Suite (EDS). The hardware mutex core allows different processors to claim ownership of a shared resource for a period of time. This temporary ownership of a resource by a processor protects the shared resource from corruption by the actions of another processor.

To prevent corruption, you must write software that waits to acquire the mutex before it accesses the shared resource, ensuring mutually exclusive access. An atomic test-and-set operation, which cannot be interrupted, allows a processor to check for ownership and acquire ownership in a single operation, avoiding the potential pitfall of two processors each confirming that no processor currently has ownership, followed by both processors acquiring the resource, violating mutual exclusion. The fact that the operation cannot be interrupted also ensures that an operating system task switch cannot occur while the processor is acquiring or releasing the mutex.

The hardware mutex core provides a semaphore for mutually exclusive access to any resource. The software determines that resource and is responsible for using the mutex core to ensure mutually exclusive access.

The hardware mailbox core allows different processors to coordinate with each other by providing both mutually exclusive access and data exchange in a single resource. The mailbox hardware core implementation includes the functionality of two mutex cores, one to guarantee mutually exclusive access for reading from the mailbox and one to guarantee mutually exclusive access for writing to the mailbox.

For more information about mutually exclusive access to shared memory, refer to "The Hardware Mutex and Mailbox Cores" on page 1–7.

The Nios II SBT for Eclipse supports software debug on multiprocessor systems, by allowing users to launch and stop software debug sessions on simultaneously running processors.

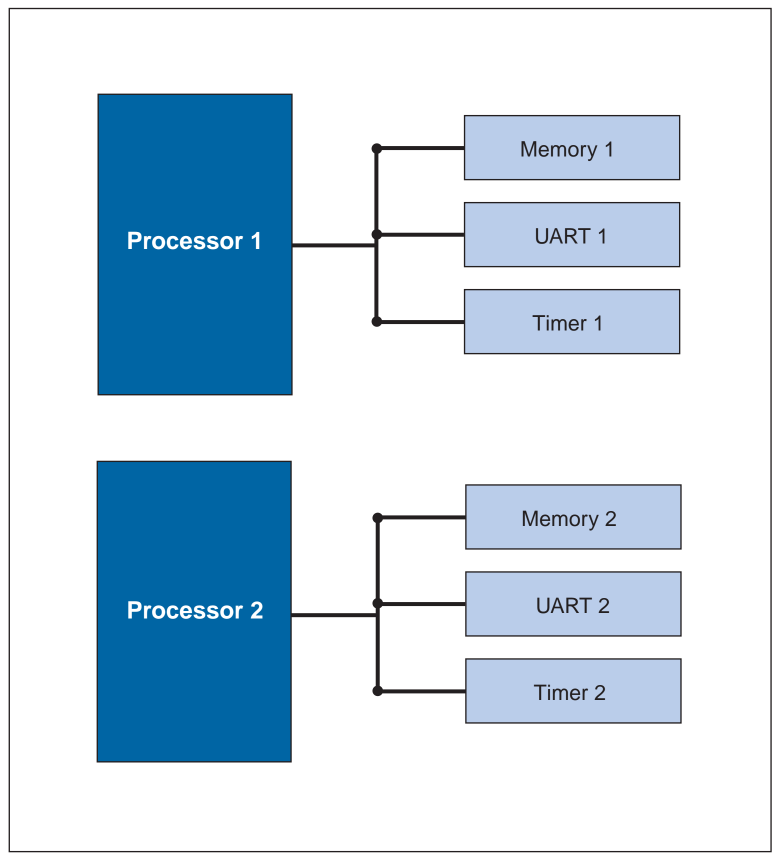
Hardware Design Considerations

Nios II multiprocessor systems are split into two main categories, those that share resources, and those in which each processor is autonomous and does not share resources with other processors.

Autonomous Multiprocessors

While autonomous multiprocessor systems contain multiple processors, these processors are completely autonomous and do not communicate with the others, much as if they were completely separate systems. Systems of this type are typically less complicated and pose fewer challenges because by design, they do not share resources and so the system's processors are incapable of interfering with each other's operation. Figure 1–1 shows a block diagram of two autonomous processors in a multiprocessor system.

Figure 1–1. Autonomous Multiprocessor System



Multiprocessors that Share Resources

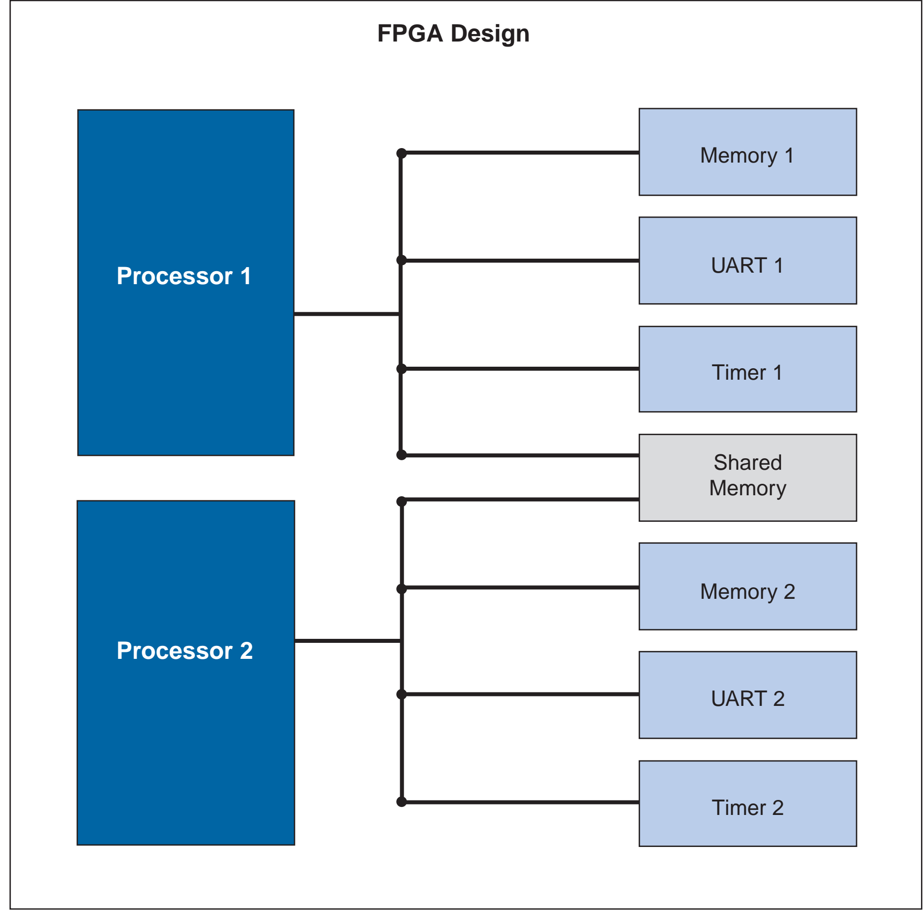
Multiprocessor systems that share resources can pose many more challenges. While the Nios II EDS includes features making it possible to reliably implement multiprocessor systems that share resources, the creation of such systems is not necessarily a straightforward venture. Altera recommends that you complete this tutorial and fully understand its recommendations before attempting to create a resource-sharing multiprocessor system.

Sharing Resources in a Multiprocessor System

Resources are considered shared when they are available to be accessed by more than one processor. The SOPC Builder connections panel controls which hardware components can be accessed by each individual Nios II processor.

Shared resources can be a very powerful aspect of multiprocessor systems, but care must be taken when deciding which system resources are shared, and how the different processors will cooperate regarding the use of resources. Figure 1–2 shows a block diagram of a sample multiprocessor system in which two processors share an on-chip memory.

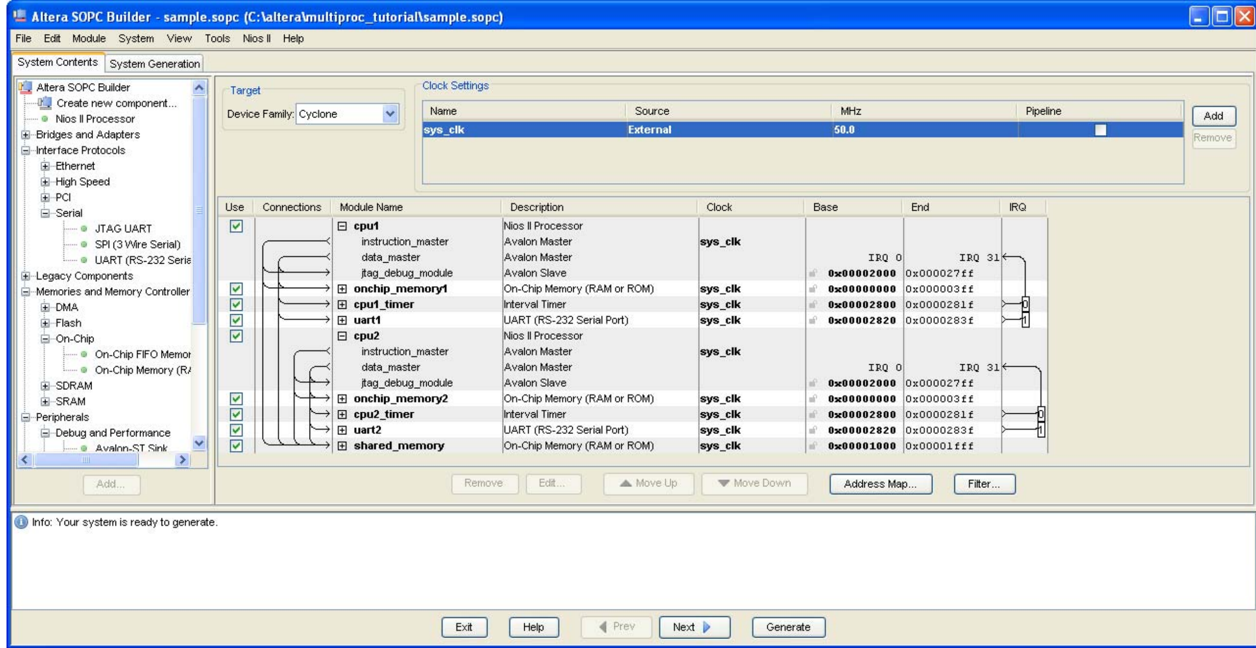
Figure 1–2. Multiprocessor System with Shared Resource



Resources can be made shareable by simply connecting them to multiple processor bus masters in the connection matrix of SOPC Builder, but that in no way guarantees that the processors that share them will do so non-destructively. The software running on each processor is responsible for coordinating mutually exclusive access to shared resources with the system's other processors.

Figure 1–3 shows a sample multiprocessor system in SOPC Builder. The component listed at the bottom, **shared_memory**, is considered shared because the data and instruction master ports of both processors are connected to the same slave port of the memory. Because **cpu1** and **cpu2** are both physically capable of writing blocks of data to the shared memory at the same time, the software for those processors must be written carefully to protect the integrity of the data stored in the shared memory.

Figure 1–3. Multiprocessor System Sharing On-Chip Memory



Sharing Memory

The most common type of shared resource in multiprocessor systems is memory. Shared memory can be used for anything from a simple flag whose purpose is to communicate status between processors, to complex data structures that are collectively computed by many processors simultaneously.

If a memory component is to contain the program memory for more than one processor, each processor sharing the memory is required to use a separate area for code execution. The processors cannot share the same area of memory for program space. Each processor must have its own unique `.text`, `.rodata`, `.rwdata`, `.heap`, and `.stack` sections. See "Software Design Considerations" on page 1–10 for information on how to make sure each processor sharing a memory component for program space uses a dedicated area within that memory.

If a memory component is to be shared for data purposes, its slave port must be connected to the data masters of the processors that are sharing the memory. Sharing data memory between multiple processors can be trickier than sharing instruction memory because data memory can be written to as well as read. If one processor is writing to a particular area of shared data memory at the same time another processor is reading or writing to that area, data corruption will likely occur, causing application errors at the very least, and possibly a system crash.

The processors sharing memory need a mechanism to inform one another when they are using a shared resource, so the other processors do not interfere.

The Hardware Mutex and Mailbox Cores

The Nios II processor provides protection of shared resources with its hardware mutex and mailbox core features. The hardware **mutex** and **mailbox** cores are not internal features of the Nios II processor; they are small SOPC Builder components.

The term mutex stands for mutual exclusion, and a mutex does exactly as its name suggests. A mutex allows cooperating processors to agree that one of them should be allowed mutually exclusive access to a hardware resource in the system. This is useful for the purpose of protecting resources from data corruption that can occur if more than one processor attempts to use the resource at the same time.

The mutex core acts as a shared resource, providing an atomic test-and-set operation that allows a processor to test if the mutex is available and if so, to acquire the mutex lock in a single operation. When the processor is finished using the shared resource associated with the mutex, the processor releases the mutex lock. Now another processor may acquire the mutex lock and use the shared resource. Without the mutex, this kind of function would normally require the processor to execute two separate instructions, test and set, between which another processor could also test for availability and succeed. This situation would leave two processors both thinking they successfully acquired mutually exclusive access to the shared resource when clearly they did not.

It is important to note that the mutex core does not physically protect resources in the system from being accessed at the same time by multiple processors. The software running on the processors is responsible for abiding by the rules. The software must be designed to always acquire the mutex before accessing its associated shared resource.

For more information about the hardware mutex core, refer to the *Mutex Core* chapter in Volume 5: Embedded Peripherals of the *Quartus II Handbook.*

The mailbox core provides safe access to the shared memory location of the mailbox messages, but only if all accesses to this shared memory are performed using the hardware mailbox core's software API. Physically, as seen in the SOPC Builder connections panel, multiple processors are connected to the shared memory through their data master ports, and in theory any one of them could simply write to the shared memory, violating mutual exclusion and overwriting data. However, if the mailbox core uses a dedicated shared memory, as in the design example for this tutorial, and all accesses to that shared memory are handled through the API, mutually exclusive access is ensured. Each hardware mailbox core you implement in your SOPC Builder system is associated with a specific shared memory.

The mailbox hardware core implementation includes the functionality of two mutex cores, one to guarantee mutually exclusive access for reading from the mailbox and one to guarantee mutually exclusive access for writing to the mailbox. Manipulation of mutex acquisition and relinquishing is handled through the API. The mailbox operations visible to the software are `altera_avalon_mailbox_pend()`, `altera_avalon_mailbox_post()`, and a non-blocking `get` operation.

For more information about the hardware mailbox core, refer to the *Mailbox Core* chapter in Volume 5: Embedded Peripherals of the *Quartus II Handbook*.

Another kind of mutex, called a software mutex, is common in many operating systems for providing the same protection of resources. The difference is that a software mutex is purely a software construct that is used to protect hardware resources from being corrupted by multiple processes running on the same processor. A hardware mutex core is an SOPC Builder component with an Avalon interface that uses logic to guarantee only one processor is granted the lock of the mutex at any given time. Similarly, a hardware mailbox core in an SOPC Builder system guarantees only one processor acquires the mailbox-associated mutex for writing at any given time, and that only one processor acquires the mailbox-associated mutex for reading at any given time. If every processor waits until it acquires the appropriate mutex before using the associated shared resource, the resource is protected from potential corruption caused by simultaneous access by multiple processors. In the case of the hardware mailbox core, the API call tells the mailbox to acquire the mutex on behalf of the processor. In the case of the hardware mutex core, each Nios II processor must acquire and relinquish the mutex from the mutex core directly. The hardware mutex core itself has no connection to the shared resource; it merely provides a semaphore. Software must be written so that no processor attempts to access the shared resource without first acquiring the mutex.

In some limited cases a mutex core or mailbox core might not be necessary. Such cases might include one-way or circular message buffer arrangements in which only one processor ever writes to a particular set of memory locations. However, sharing resources safely without a mutex core or mailbox core can be complicated. When in doubt, use the mutex core or the mailbox core.

Sharing Peripherals Between Multiple Processors

In general, with the exception of the mutex core and the mailbox core, the Nios II EDS does not support sharing non-memory peripherals between multiple processors.

Sharing peripherals in multiprocessor systems presents some difficult challenges, and is generally considered to lead to inefficient system designs. The biggest problems arise for peripherals with interrupts. If a peripheral is allowed to interrupt all the processors that share it, there is no reliable way to guarantee which processor will respond first and service that interrupt. Additionally, if the peripheral is used as an input device for multiple processors, it becomes difficult to determine which processor is supposed to collect given input from the device. While it is conceivable that a complex system of handshaking could be created to handle these scenarios, such a system is beyond the scope of this document, and is unsupported by the Nios II hardware abstraction layer (HAL) library.

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

Memory peripherals and multiprocessor coordination peripherals can be accessed by multiple processors. Altera recommends that you restrict all other peripherals to be accessible by only one processor in the system. If other processors require use of the peripheral, they should use a hardware mailbox, or a message buffer that is mutex-protected or otherwise multiprocessor-safe, to communicate with the single processor that is connected to that peripheral.

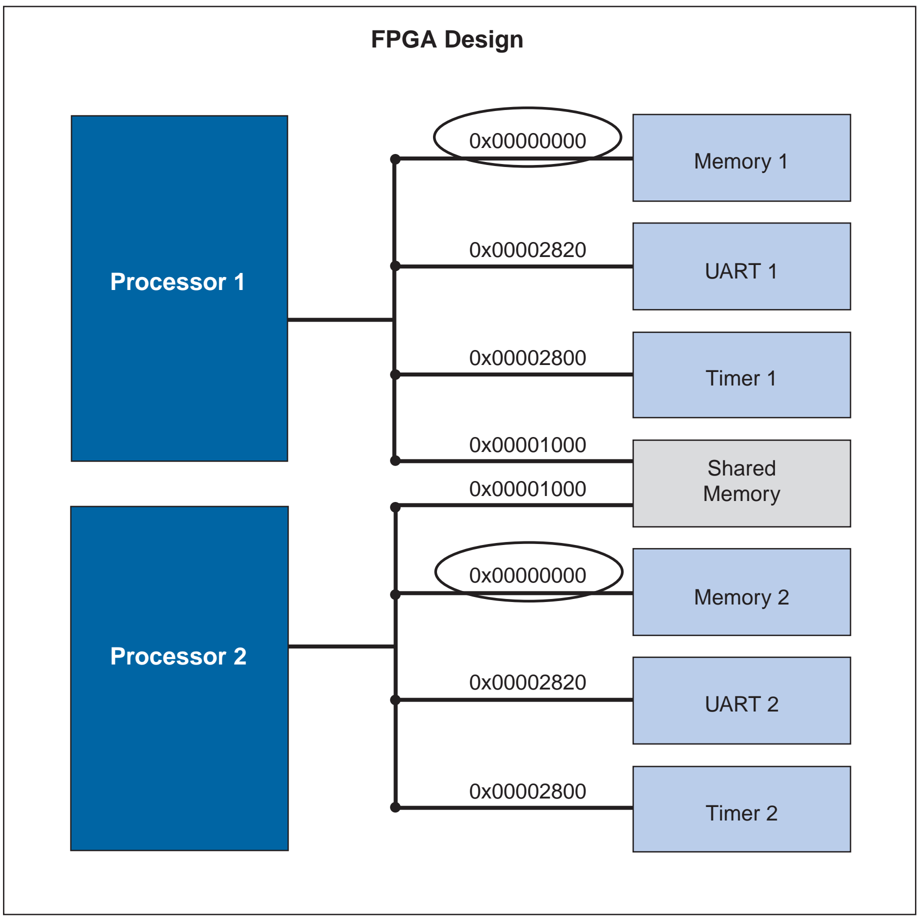
When building any system, especially a multiprocessor system, it is advisable to only make connections between peripherals that require communication. For instance, if a processor runs from and uses only one on-chip memory, there is no need to connect that processor to any other memory in the system. Physically disconnecting the processor from memories it is not using both saves FPGA resources and guarantees the processor will never corrupt those memories.

In single processor systems, SOPC Builder will usually make intelligent default choices for connecting components. However, in multiprocessor systems the need to connect different components is very design dependent. Therefore, when designing multiprocessor systems, you should explicitly verify that each component is connected to the desired processor. Most components should be managed by a single processor. If CPU A requires the services of a peripheral that is connected to and managed by CPU B, CPU A must request of CPU B that it perform operations with the peripheral on behalf of CPU A. You can use mailbox core communication between the two processors for this purpose.

Multiprocessors and Overlapping Address Space

Single-processor systems typically prohibit more than one slave peripheral from occupying the same address space because this arrangement causes conflicts. In multiprocessor systems however, separate slave peripherals can occupy the same base address and not conflict, as long as each of those peripherals is exclusively mastered by a different processor. Because not every slave peripheral is necessarily mastered by every processor, each processor might have a different view of the system. If processor A is connected to a slave peripheral mapped to address `0x4000`, processor B may connect to a separate slave peripheral, also mapped to address `0x4000`, as long as processor A is not connected to processor B's slave peripheral and processor B is not connected to processor A's slave peripheral. In effect, the point-to-point connectivity allows the two processors to have separate address spaces. Figure 1–4 shows a block diagram of a sample multiprocessor system with different slave components mapped to the same base address.

Figure 1–4. Multiprocessor Slave Peripherals Mapped to the Same Base Address



Software Design Considerations

Creating and running software on multiprocessor systems is much the same as for single-processor systems, but requires the consideration of a few additional points. Many of the software design issues described in this section are dictated by the system's hardware architecture.

Program Memory

When creating multiprocessor systems, you might want to run the software for more than one processor out of the same physical memory device. Software for each processor must be located in its own unique region of memory, but those regions are allowed to reside in the same physical memory device. For instance, imagine a two-processor system where both processors run out of SDRAM. The software for the first processor requires 128 KBytes of program memory, and the software for the second processor requires 64 KBytes. The first processor could use the region between `0x0` and `0x1FFFF` in SDRAM as its program space, and the second processor could use the region between `0x20000` and `0x2FFFF`.

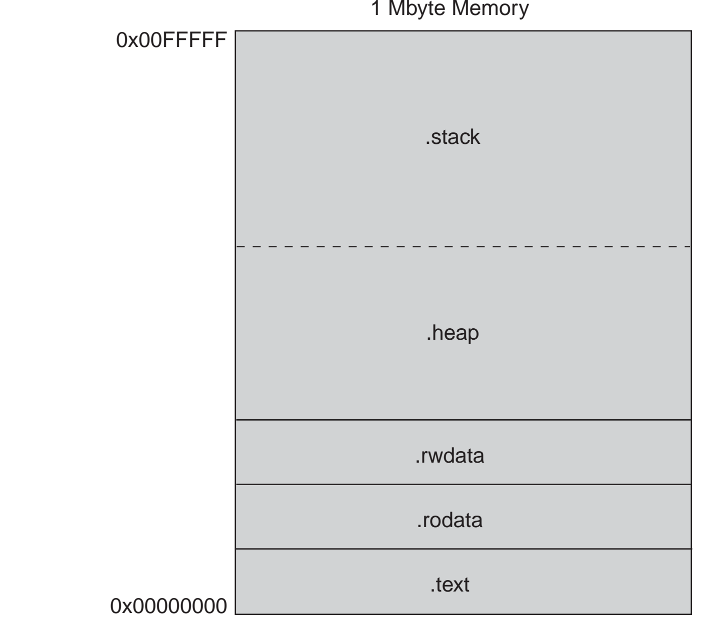
The Nios II SBT and SOPC Builder provide a simple scheme of memory partitioning that allows multiple processors to run their software out of different regions of the same physical memory. The partitioning scheme uses the exception address for each processor, which is set in SOPC Builder, to determine the region of memory from which each processor will be allowed to run its software. The Nios II SBT is ultimately responsible for the linking of the processors' software and determining where the software will reside in memory; it uses the exception addresses that were set for each processor in SOPC Builder to calculate where the different code sections will be linked. The Nios II SBT provides each processor its own section within memory from which it can run its software. If the software for two different processors is linked to the same physical memory, then the exception address of each processor is used to determine the base address of the region which that processor's software can occupy. The end address of the region is determined by the next exception address found in that physical memory, or the end of that physical memory, whichever comes first.

Each processor in a single or multiprocessor system has five primary code sections that need to be linked to fixed addresses in memory. These sections are:

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

See Figure 1–5 for a memory map showing how these sections are typically linked in memory for a single processor Nios system.

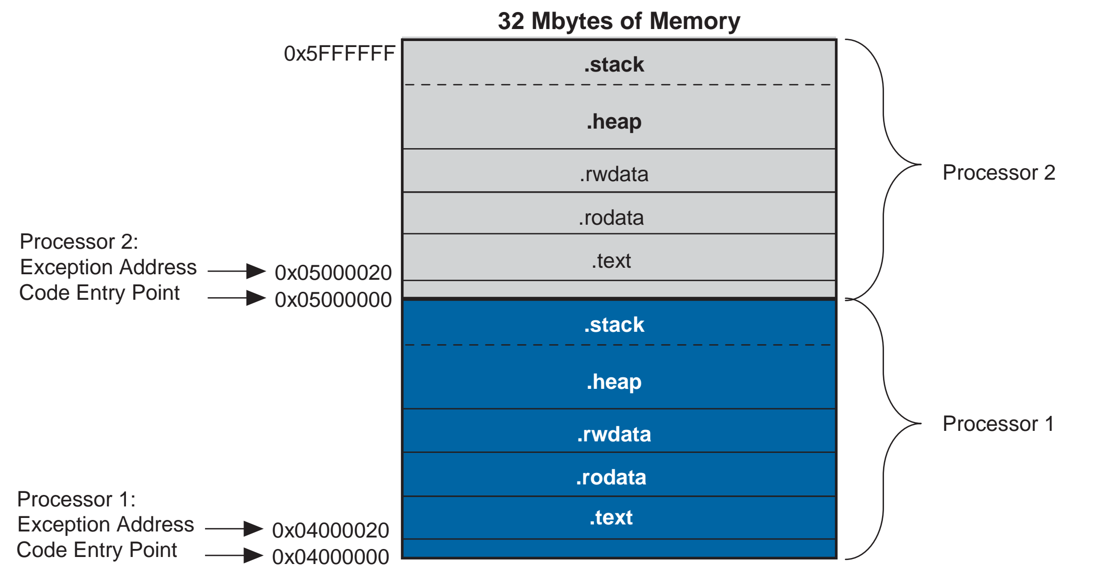
Figure 1–5. Single Processor Code Linked in Memory Map



In a multiprocessor system, it might be advantageous to use a single memory to store all the code sections for each processor. In this case, the exception address set for each processor in SOPC Builder is used to define the boundaries between where one processor's code sections end and where the next processor's code sections begin.

For instance, imagine a system where SDRAM occupies the address range `0x04000000–0x5FFFFFF` and processors A and B are each allocated 16 MBytes of SDRAM to run their software. If you use SOPC Builder to set their exception addresses 16 MBytes apart in SDRAM, the Nios II SBT automatically partitions SDRAM based on those exception addresses. See Figure 1–6 for a memory map showing how the SDRAM is partitioned in this example system.

Figure 1–6. Partitioning of SDRAM Memory Map for Two Processors



The lower six bits of the exception address are always set to `0x20`. Offset `0x0` is where the Nios II processor must run its reset code, so the exception address must be placed elsewhere. The offset of `0x20` is used because it corresponds to one instruction cache line. The `0x20` bytes of reset code initialize the instruction cache, and then branch around the exception section to the system startup code.

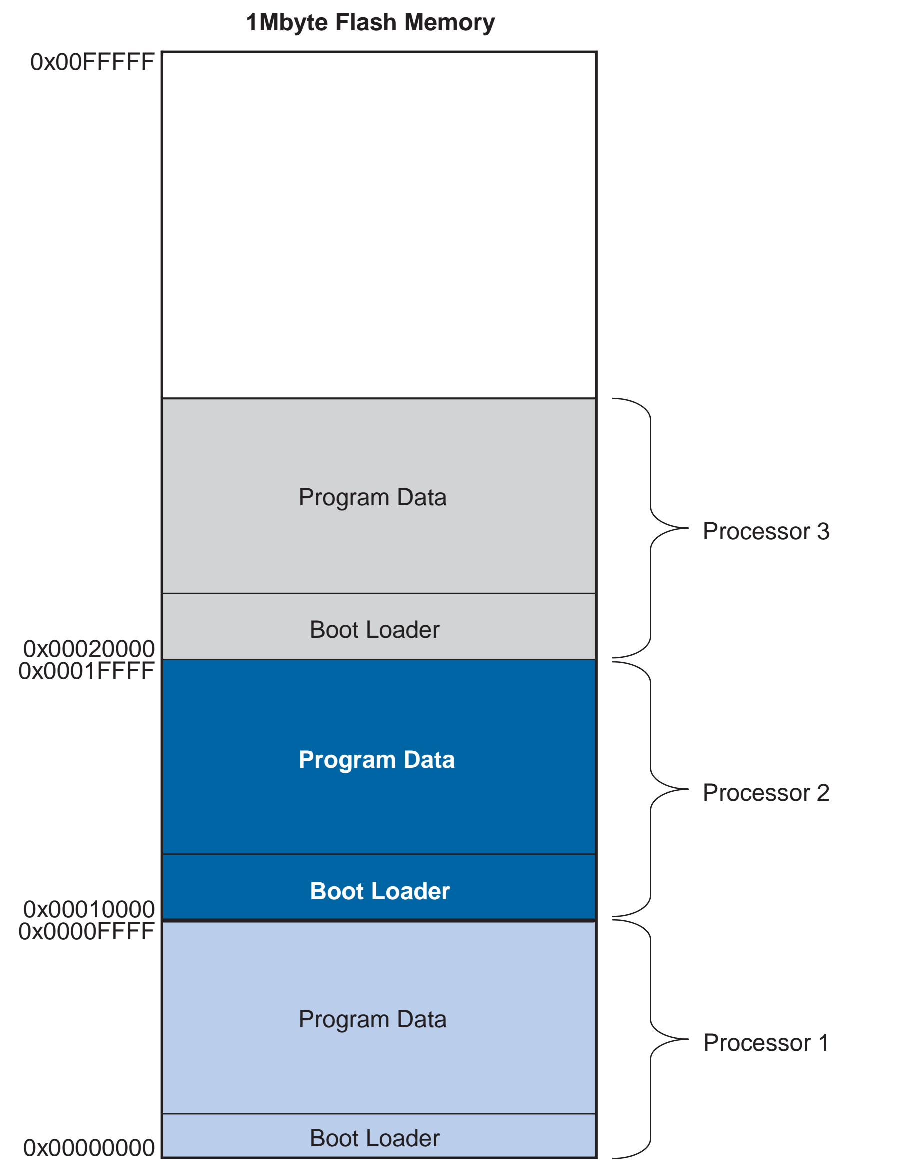
Care must be taken when partitioning a physical memory to contain the code sections of multiple processors. There are no safeguards in SOPC Builder or the Nios II SBT that guarantee you have provided enough code space for each processor's stack and heap in the partition. If inadequate code space is allotted in memory, the stack and heap might overflow and corrupt the processor's code execution.

Boot Addresses

In multiprocessor systems, each processor must boot from its own piece of memory. Multiple processors might not boot successfully from the same bit of executable code at the same address in the same non-volatile memory. Boot memory can also be partitioned, much like program memory can, but the notion of sections and linking is not a concern as boot code typically just copies the real program code to where it has been linked in RAM, and then branches to the program code. To boot multiple

processors out of separate regions with the same non-volatile memory device, simply set each processor's reset address to the location from where you wish to boot that processor. Be sure you leave enough space between boot addresses to hold the intended boot payload. See Figure 1–7 for a memory map of one physical flash device from which three processors can boot.

Figure 1–7. Flash Device Memory Map with Three Processors Booting



The Nios II Flash Programmer is able to program bootable code for multiple processors into a single flash device. The flash programmer looks at the reset address of each processor and uses that reset address to calculate the offset within the flash memory where the code is programmed.

For details about the Nios II Flash Programmer, refer to the *Nios II Flash Programmer User Guide*.

You must exercise caution when connecting multiple Nios II processors to a single CFI flash memory device. Because no support mechanism exists in the CFI flash driver to allow a processor to confirm that another processor is not currently accessing the flash memory device, a read operation can return corrupted data. Specifically, if a processor attempts to read from a CFI flash memory device currently not in read mode, the read operation does not access the data on the flash correctly. If another processor issues a query to the flash memory device immediately prior to the first processor's read attempt, the flash memory device is in command mode while it processes the query, and the read operation cannot read the data correctly.

An example of a multiprocessor system in which this caution is relevant, is a system in which multiple processors boot from the same CFI flash memory. In such systems, you must remove CFI flash memory initialization from the `alt_main()` function run by every Nios II processor. Should you find it necessary for a single processor to initialize CFI flash memory, you must ensure that it does so only after all the other processors have completed their boot processes. Otherwise, if the processors boot simultaneously, a race condition occurs when the first processor to jump out of the boot copier and start running the application code runs `alt_main()` and initializes the flash memory driver while another processor is still trying to read its own boot code. During this initialization, the second processor's read operations of its own boot copier return corrupted data.

Altera recommends that you designate one Nios II processor as the flash master, and allow only the flash master to read from or write to the flash memory device, in any system that connects multiple Nios II processors to a single flash memory device. The designated processor can read the application images from the flash memory device for the other processors.

If you choose to allow multiple Nios II processors to boot from the same CFI flash memory device, to ensure safe access to the CFI flash memory, you must remove the CFI flash memory driver initialization from the `alt_main()` function for all but one processor, and that processor must confirm boot completion by all the other processors before proceeding with the CFI flash memory driver initialization.

For information about complex boot procedures, refer to *AN458: Alternative Nios II Boot Methods*.

Running and Debugging Multiprocessor Systems from the Nios II SBT for Eclipse

The Nios II SBT for Eclipse includes a number of features that can help in the development of software for multiprocessor systems. Most notable is the ability of the Nios II SBT for Eclipse to perform simultaneous debug for multiple processors. Multiple debug sessions can run at the same time on a multiprocessor system and can pause and resume each processor independently. Breakpoints can also be set individually per processor. If one processor hits a breakpoint, it does not halt or affect the operation of the other processors. Debug sessions can be launched and stopped independently.

Design Example

The following exercise shows you how to build a two-processor Nios II system with SOPC Builder, starting with the **neek_vic_single_91sp1_v1** example design as a template. You create two application projects and two BSP software projects and import them to the Nios II SBT for Eclipse, one BSP project for each processor. The software for **cpu1** runs a TCP/IP stack and accepts commands for adjusting the frequency of a blinking LED through a telnet connection. **cpu1** uses the hardware mailbox component to pass LED frequency change messages to **cpu2**. The software for **cpu2** uses the hardware mailbox core to read these messages and adjust the LED frequency accordingly. **cpu2** continually checks the mailbox for new messages, and if it finds one, adjusts the LED frequency.

Hardware and Software Requirements

To use this design example you must have the following:

- Quartus® II Software version 9.1 SP1 or higher
- Nios II Embedded Evaluation Kit (NEEK) with the following connections:
  - Connected through a USB-Blaster connection to the host computer
  - Connected through an Ethernet cable to the network

If you do not have a NEEK, you can follow the hardware development steps, but you will not be able to download the complete system to a working board.

Installation Notes

For installation notes specific to Altera software versions, refer to the **readme.txt** file included in your **Multiprocessor_Tutorial.zip** installation.

Creating the Hardware System

In the following steps you create a multiprocessor system by starting with the **neek_vic_single_91sp1_v1** hardware example design available with this tutorial in **Multiprocessor_Tutorial.zip**, and adding an additional processor, an additional timer, an additional vectored interrupt controller, a hardware mailbox component, a shared on-chip memory, and a hardware mutex component. Your final system should be identical to that in the **neek_vic_multi_91sp1_v1** hardware design available with this tutorial in **Multiprocessor_Tutorial.zip**, for comparison purposes. If you do not have a NEEK, you can still follow these steps to learn how to design multiprocessor hardware.

Getting Started with the neek_vic_single_91sp1_v1 Example Design

To begin building a multiprocessor system sharing resources, perform the following steps:

1. Unzip the **Multiprocessor_Tutorial.zip** file.
2. Copy the **neek_vic_single_91sp1_v1** folder to a working directory of your choice. Make sure the path name has no spaces.
3. Open the Quartus II software.

4. On the File menu, click **Open Project** (not Open).
5. Browse and load the Quartus II Project File (**.qpf**) from the newly-created directory.
6. On the Tools menu, click **SOPC Builder**.
7. Click **cpu1_vic**.
8. Click the blue arrow Move Up button several times to move **cpu1_vic** directly under **cpu1**.

In this tutorial, you must name the hardware components exactly according to the instructions. If your component names differ from the names printed here, the software example will not work.

Adding a Second Processor

In the next series of steps, you add a second Nios II processor to the system. You use a Nios II/f processor because it is the fastest choice. If your FPGA is resource constrained by your other application needs, you can use the smaller Nios II/s processor.

To add a second processor, perform the following steps:

1. In the Component Library on the left side of the **System Contents** tab, expand **Processors**, and select **Nios II Processor.**
2. Click **Add.** The Nios II Processor MegaWizard™ interface appears, displaying the **Core Nios II** page.
3. Specify the settings shown in Table 1–1.

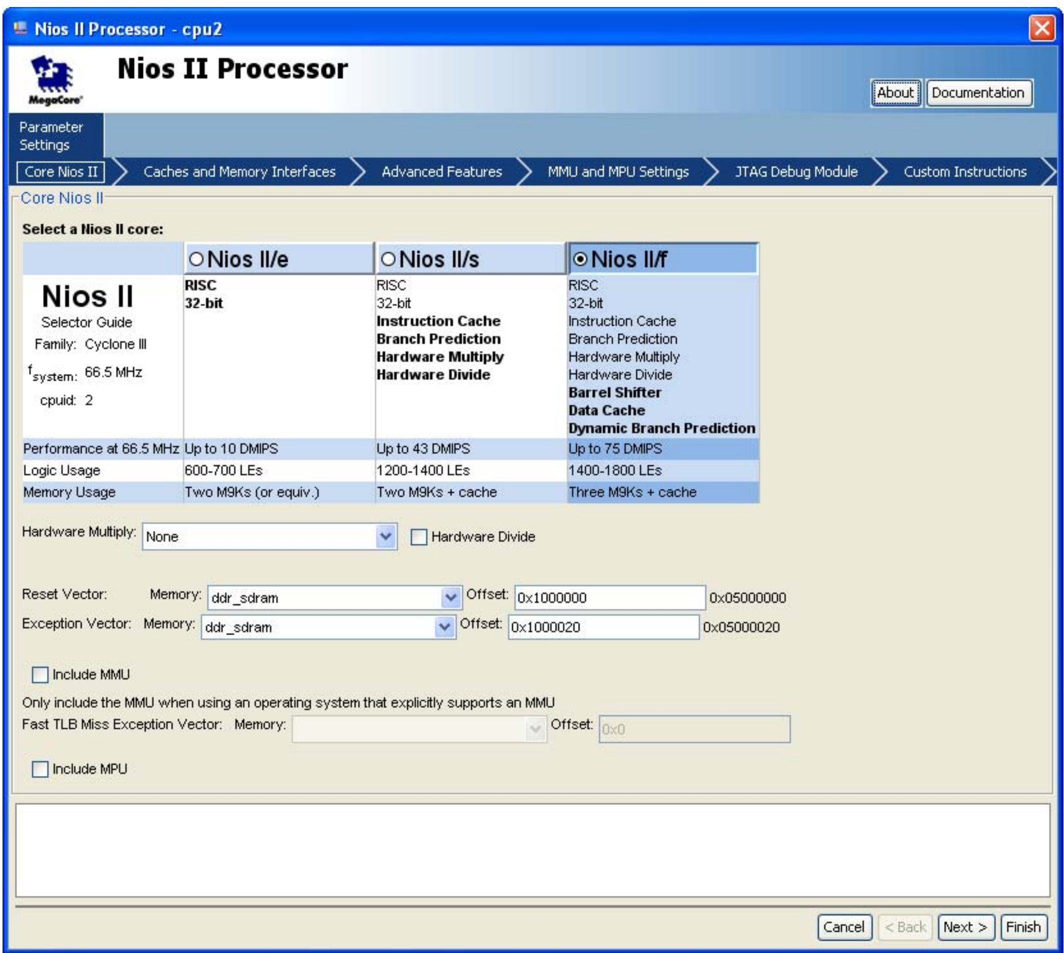
Table 1–1. cpu2 Parameter Settings

Parameter	Value
Nios II Core	**Nios II/f**
Hardware Multiply	**None**
Hardware Divide	Off
Reset Vector: Memory	**ddr_sdram**
Reset Vector: Offset	`0x1000000`
Exception Vector: Memory	**ddr_sdram**
Exception Vector: Offset	`0x1000020`
Include MMU	Off
Include MPU	Off

Recall from “Program Memory” on page 1–10 that the exception addresses determine how code memory is partitioned between processors. In this tutorial, each of the two processors runs its software from 16 Mbyte of SDRAM, so you set each processor's exception address within SDRAM, separated by 0x1000000 (16 MBytes).

Figure 1–8 shows the **Core Nios II** page after you specify these settings.

Figure 1–8. Nios II Processor Settings for cpu2



4. Click **JTAG Debug Module**. The **JTAG Debug Module** page appears.
5. Select **Level 1** as the debugging level for this processor.
6. Click **Advanced Features**. The **Advanced Features** page appears.
7. For **Interrupt Controller**, select **External**.
8. For **Number of shadow register sets,** select **7**.
9. Turn on **Assign cpuid control register value manually**.
10. For **cpuid control register value,** type `0x2`.
11. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the Nios II core named **cpu_0** now appears at the bottom of the System Contents description.
12. Right-click the newly-added processor **cpu_0** and click **Rename**.
13. Type `cpu2` and press Enter.
14. In the **Clock** column, double-click and select **ddr_sdram_auxhalf**.

15. In the **Base** column, double-click and type `0x06020000`.
16. Click the blue arrow Move Up button several times to move **cpu2** directly under **cpu1_timer**.

Error messages still appear in the SOPC Builder messages window. This is because SOPC Builder does not know that you plan to connect this processor with other components in the system. Ignore the error messages for now. You will fix these errors in later steps.

Adding a Vectored Interrupt Controller for cpu2

Processors in an SOPC Builder system should not share a vectored interrupt controller (VIC). Not every processor requires a VIC; however, multiple processors cannot share a VIC. A VIC sends peripheral interrupts to the processor to which it is connected. To protect data integrity, each peripheral interrupt must be handled by a single processor.

SOPC Builder allows you to connect a peripheral to more than one VIC. You must check your system manually to ensure that each peripheral interrupt is routed to no more than one VIC.

To add a VIC for **cpu2**, perform the following steps:

1. In the Component Library, expand **Processor Additions** and then click **Vectored Interrupt Controller**.
2. Click **Add**. The VIC MegaWizard interface appears.
3. Specify the following settings:
   - **Number of Interrupts:** 8
   - **Requested Interrupt Level Width:** 4
   - **DAISY CHAIN ENABLE:** Off
4. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the VIC named **vic_0** now appears at the bottom of the System Contents description.
5. Right-click **vic_0** and click **Rename**.
6. Type `cpu2_vic` and press Enter.
7. In the **Clock** column, double-click and select **ddr_sdram_auxhalf**.
8. In the **Base** column, double-click and type `0x06021000`.
9. Click the blue arrow Move Up button several times to move **cpu2_vic** directly under **cpu2**.
10. Using the connection matrix, make the following port connections:
    - **cpu2_vic/dummy_master** to **cpu2_vic/csr_access**
    - **cpu2/data_master** to **cpu2_vic/csr_access**
    - **cpu2_vic/interrupt_controller_out** to **cpu2/interrupt_controller_in**

If you do not see the connection matrix, on the View menu, click **Show Connections Column**.

Adding a Timer for cpu2

As mentioned earlier, it is typically not recommended for multiple processors to share non-memory or non-multiprocessor coordination peripherals, so in this section you add a separate timer peripheral for the second processor in this system.

To add a timer for **cpu2**, perform the following steps:

1. In the Component Library, expand **Peripherals**, expand **Microcontroller Peripherals**, and then click **Interval Timer**.
2. Click **Add**. The Interval Timer MegaWizard interface appears.
3. Specify the following settings:
   - **Timeout period:** 1 ms
   - **Timer counter size:** 32
   - Under **Hardware Options**, in the **Presets** list, select **Full-featured**.
4. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the interval timer named **timer_0** now appears at the bottom of the System Contents description.
5. Right-click **timer_0** and click **Rename**.
6. Type `cpu2_timer` and press Enter. This is the timer for **cpu2**.
7. In the **Clock** column, double-click and select **ddr_sdram_auxhalf**.
8. In the **Base** column, double-click and type `0x06022000`.
9. Click the blue arrow Move Up button to move **cpu2_timer** directly under **cpu2_vic**.
10. In the connection matrix, connect **cpu2_timer/s1** to **cpu2/data_master** only, and disconnect **cpu2_ timer** from all other masters.
11. In the IRQ connection matrix (on the rightmost side of the **System Contents** tab), type `0` at the **cpu2_vic** to **cpu2_timer** connection. This value allows **cpu2_timer** to interrupt **cpu2** with a priority setting of 0, which is the highest priority. Do not specify any interrupt priority for the **cpu1_vic**, because the **cpu2_timer** should only interrupt the **cpu2** processor.

   If your system contents are displayed using the default filter, the labels you see when you move the mouse over the potential interrupt connections in the IRQ column do not make sense. Despite the port names listed, the ports that are connected in the IRQ column are the **irq_input** and **irq** ports of the components. To see the correct labels, in the **System Contents** tab, click the Filters button and for **Filter,** select **All**. To return to the more compressed presentation, in the **System Contents** tab, click the Filters button and for **Filter**, select **Default**. Whether or not you display the correct labels, the correct ports are connected in the IRQ column.

Adding a Message Buffer Memory

In this section, you add an on-chip memory to the system. The hardware mailbox component uses this memory as a message buffer to pass messages between processors. This memory must be shared by both processors in the system. The processors use the mailbox core to provide mutually exclusive memory access, protecting the memory's contents from corruption.

To add a message buffer memory, perform the following steps:

1. In the Component Library, expand **Memories and Memory Controllers**, expand **On-Chip**, and then click **On-Chip Memory (RAM or ROM).**
2. Click **Add.** The On-Chip Memory (RAM or ROM) MegaWizard interface appears.
3. Under **Memory type**, select **RAM (Writable)**.
4. For **Data width**, select **32.**
5. In the **Total memory size** box, type `4096` and select **Bytes** to specify a memory size of 4 KBytes.
6. Under **Read latency**, for **Slave s1**, select **1**.
7. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the on-chip memory named **onchip_memory2_0** now appears at the bottom of the System Contents description.
8. Right-click **onchip_memory2_0** and click **Rename**.
9. Type `message_buffer_ram` and press Enter. This memory is used as a message buffer for the two processors in your multiprocessor system to communicate through the hardware mailbox component.
10. In the **Clock** column, double-click and select **ddr_sdram_auxhalf**.
11. In the **Base** column, double-click and type `0x07000000`.
12. Click the blue arrow Move Up button to move **message_buffer_ram** directly under **led_pio.**

Connecting Shared Memory and System Output Resources

Now you need to properly connect all the resources that are shared between processors in the system using SOPC Builder's connection matrix and IRQ connection matrix.

To properly connect the memory and system output resources in the system shared by the multiple processors, perform the following steps:

1. In the connection matrix, ensure that each timer is connected only to the data master for its processor, by disconnecting **cpu1_timer** from **cpu2/data_master**.
2. In the connection matrix, ensure that the System ID peripheral named **sysid** is connected to the data masters for both processors.
3. In the connection matrix, connect **message_buffer_ram** to the data masters for both processors.
4. In the connection matrix, ensure that the **ddr_half_rate_bridge** is connected to the instruction and data masters for each processor, allowing both processors to access **ddr_sdram**.

5. In the connection matrix, disconnect **cpu2/instruction_master** from all components except **cpu2/jtag_debug_module**, **ddr_half_rate_bridge**, and **flash_tristate_bridge**.
6. In the connection matrix, disconnect **cpu2/data_master** from all components except **cpu2/jtag_debug_module**, **sysid**, **cpu2_vic/csr_access**, **cpu2_timer**, **led_pio**, **message_buffer_ram**, **ddr_half_rate_bridge**, and **flash_tristate_bridge**.
7. In the IRQ connection matrix, ensure that **cpu2_vic** is connected only to the **cpu2_timer**. To remove an IRQ connection, erase the default IRQ number. In this step, you disconnect all IRQ lines involving the **cpu2_vic** from all resources, except for the **cpu2_vic** to **cpu2_timer** IRQ connection you set in step 11 in "Adding a Timer for cpu2" on page 1–19.

   ☞ Recall that to view the correct port names for the IRQ connections, you must use the **All** filter in the **System Contents** tab.

8. In the IRQ connection matrix, ensure that **cpu1_vic** is not connected to the **cpu2_timer**.
9. In the IRQ connection matrix, connect **cpu1_vic** to all peripherals except the **cpu2_timer** by setting unique interrupt numbers for each potential **cpu1_vic** IRQ connection.
10. To design the system so that **cpu2** controls the blinking of the LED on the NEEK, perform the following steps:
    a. In the connection matrix, disconnect **cpu1/data_master** from the **led_pio** component.
    b. In the connection matrix, connect **cpu2/data_master** to the **led_pio** component.

Adding a Hardware Mailbox Component

Because your multiprocessor system shares data in memory, it must include an SOPC Builder hardware component for multiprocessor coordination to protect that memory from data corruption.

To add a hardware mailbox, perform the following steps:

1. In the Component Library, expand **Peripherals**, expand **Multiprocessor Coordination**, and then click **Mailbox**.
2. Click **Add**. The Mailbox MegaWizard interface appears.
3. Leave the **Memory module** blank.
4. For **Shared mailbox memory offset**, type `0x800`.
5. For **Mailbox size (bytes)**, type `0x100`.
6. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the mailbox named **mailbox_0** now appears at the bottom of the System Contents description.
7. Right-click **mailbox_0** and click **Rename**.
8. Type `message_buffer_mailbox` and press Enter.
9. In the **Clock** column, double-click and select **ddr_sdram_auxhalf**.

10. In the **Base** column, double-click and type `0x07009000`.
11. Click the blue arrow Move Up button to move **message_buffer_mailbox** directly under **message_buffer_ram**.

Optionally Adding a Hardware Mutex Component

The software examples for this tutorial do not use a separate hardware mutex component. Instead, they coordinate multiprocessor communication through the hardware mailbox component. However, this section provides you with the instructions to add a mutex to your system, to demonstrate its availability and to illustrate how to connect a hardware mutex component in a multiprocessor SOPC Builder system.

To add the hardware mutex peripheral, perform the following steps:

1. In the Component Library, expand **Peripherals**, expand **Multiprocessor Coordination**, and then click **Mutex.**
2. Click **Add**. The Mutex MegaWizard interface appears.
3. Click **Finish** to accept the default settings for this component. You return to the SOPC Builder **System Contents** tab, and an instance of the mutex named **mutex_0** now appears at the bottom of the System Contents description.
4. Right-click **mutex_0** and click **Rename**.
5. Type `message_buffer_mutex` and press Enter.
6. In the **Clock** column, double-click and select **ddr_sdram_auxhalf**.
7. In the **Base** column, double-click and type `0x07008000`.
8. Click the blue arrow Move Up button to move **message_buffer_mutex** directly under **led_pio**.

Connecting Shared Multiprocessor Coordination Resources

Now you need to properly connect all the coordination resources that are shared between processors in the system using SOPC Builder's connection matrix and IRQ connection matrix.

To properly connect the shared coordination resources in the multiprocessor system, perform the following steps:

1. In the connection matrix, connect **message_buffer_mutex** to the **data_master** ports of both processors.
2. Ensure that **message_buffer_mutex** is not connected to either processor’s **instruction_master** port.
3. In the connections matrix, connect **message_buffer_mailbox** to the **data_master** ports of both processors.
4. Ensure that **message_buffer_mailbox** is not connected to either processor’s **instruction_master** port.
5. Double-click the **message_buffer_mailbox** component. The Mailbox MegaWizard interface appears.
6. For memory module, select **message_buffer_ram**.

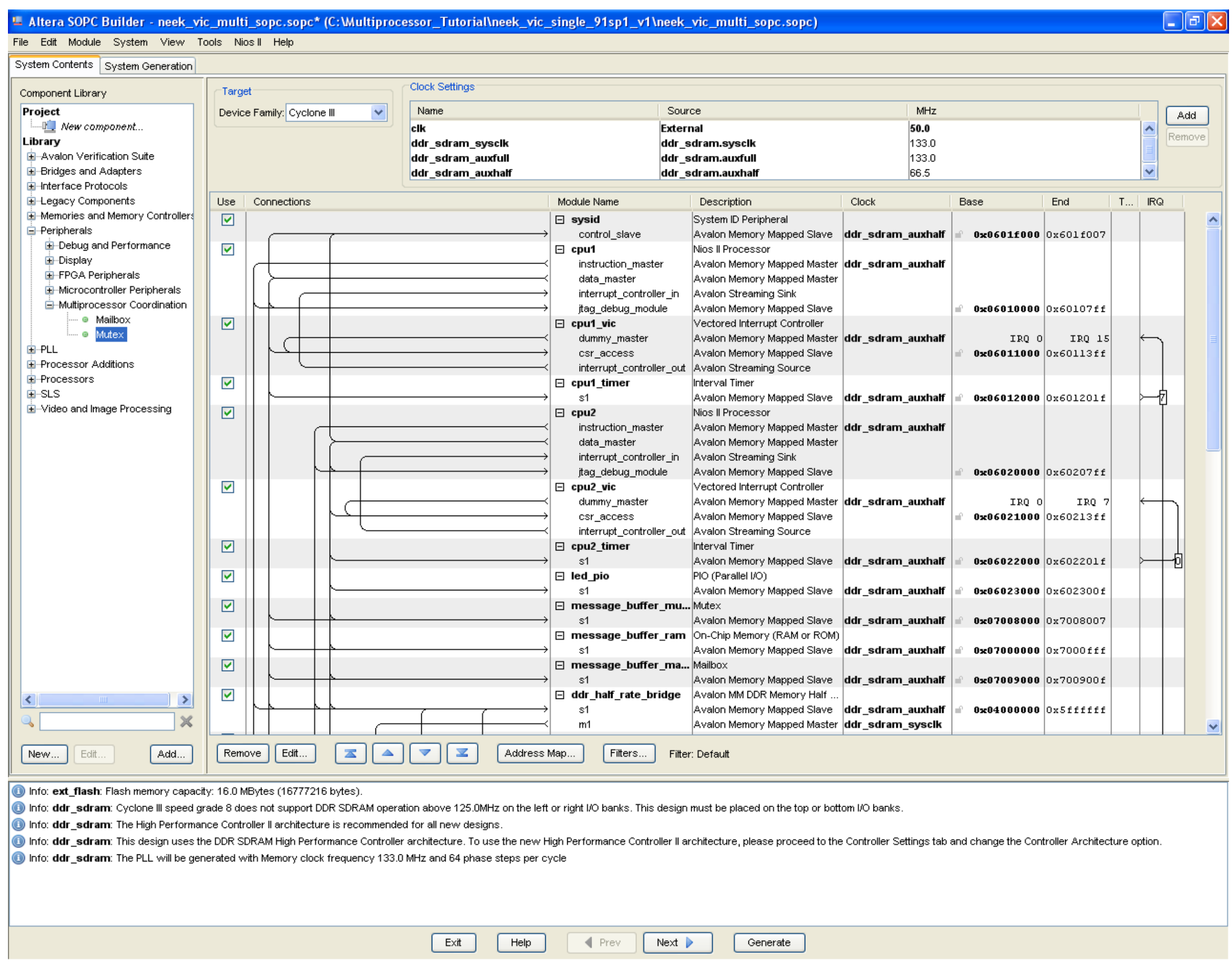
 © February 2010 Altera Corporation

7. Click **Finish**.

None of the I/O components should be connected to multiple processors. Unused connections consume Avalon resources and FPGA logic elements, possibly affecting system f_{MAX}.

Figure 1–9 shows a system in SOPC Builder after these changes. It shows the new components that implement the message buffer and the required connectivity for the system.

Figure 1–9. Shared Resource Connections



The **neek_vic_multi_91sp1_v1** example design is also available in the **Multiprocessor_Tutorial.zip** file. You can compare your completed system to the predefined system located in the **neek_vic_multi_91sp1_v1** subdirectory of the extracted directory.

Generating and Compiling the System

In this section, you generate HDL for the system you just constructed in SOPC Builder, and then compile the project in the Quartus II software to produce a programming file.

To generate and compile the system, perform the following steps:

1. Click the **System Generation** tab.
2. Turn off **Simulation. Create project simulator files.** System generation executes much faster when simulation is off.
3. Click **Generate**. This might take a few moments. A **Stop** button replaces the **Generate** button, indicating generation is taking place.
4. When generation is complete, the **Generate** button replaces the **Stop** button, and a **SUCCESS: SYSTEM GENERATION COMPLETED** message displays. Click **Exit** in SOPC Builder to return to the Quartus II software.
5. On the Quartus II Processing menu, click **Start Compilation** to compile the project in the Quartus II software.
6. When compilation completes and displays the **Full compilation was successful** message box, click **OK**.
7. Click **Programmer** on the Tools menu.
8. Turn on the **Program/Configure** checkbox for the SRAM Object File (**.sof**) in the Quartus II Programmer.
9. Click **Start** to download the FPGA configuration data to your target hardware.

Creating Software for the Multiprocessor System

In the following steps you build one application and one BSP project for each processor in the system using the Nios II SBT, and then import these application and BSP projects to the Nios II SBT for Eclipse, creating a total of four separate software projects for the multiprocessor system. You then debug the software projects using the Nios II SBT for Eclipse.

The software you run on this system uses the hardware mailbox to exchange messages between two Nios II processors.

Building the Application and BSP Projects

To build the application and BSP projects for this tutorial, perform the following steps:

1. Start a Nios II Command Shell.
2. Change directories to your working directory for the NEEK multiprocessor design example.
3. Change directories to **software_examples/app/cpu2_led**.
4. Build the project to be run on the **cpu2** processor by typing the following command:

   ```
   ./create-this-app ↵
   ```

5. Change directories to **software_examples/app/niosII_multicore_socket_server**.
6. Build the project to run on the **cpu1** processor by typing the following command:

   ```
   ./create-this-app ↵
   ```

Both application software projects and both BSP projects are created and built.

If you make hardware design changes in SOPC Builder, you can most easily regenerate the application and BSP projects by performing the following steps:

1. Delete all folders and files except the two **create-this-bsp** scripts from **software_examples/bsp/ucosii_niosII_multicore** and **software_examples/bsp/cpu2_ucosii_niosII_multicore**.
2. Delete the two files **software_examples/app/cpu2_led/Makefile** and **software_examples/app/niosII_multicore_socket_server/Makefile.**
3. Repeat the preceding set of steps to build the application and BSP projects.

Starting the Nios II SBT for Eclipse

In this section, you start the Nios II SBT for Eclipse and begin importing software projects for the two Nios II processors in the system. To start the Nios II SBT for Eclipse from SOPC Builder, perform the following steps:

1. On the Tools menu in the Quartus II software, click **SOPC Builder.**
2. In SOPC Builder, click the **System Generation** tab.
3. Click **Nios II Software Build Tools for Eclipse.** The Nios II SBT for Eclipse starts.

If the **Workspace Launcher** dialog box appears, click **OK** to accept the default workspace. If the Nios II SBT for Eclipse welcome screen appears, click **Workbench** to continue.

Importing the Software Projects

In this section, you import the following four projects to the Nios II SBT for Eclipse:

- **cpu2_led**
- **niosII_multicore_socket_server**
- **cpu2_ucosii_niosII_multicore**
- **ucosii_niosII_multicore**

To import the software projects, perform the following steps:

1. In the Nios II SBT for Eclipse, on the File menu, click **Import.**
2. Click the plus sign to the left of **Nios II Software Build Tools Project.** The **Import Nios II Software Build Tools Project** option appears.
3. Highlight **Import Nios II Software Build Tools Project,** and click **Next.** The **Import Software Build Tools Project** window opens.
4. For **Project location,** click the Browse button to navigate to one of the four projects that you built by running the **create-this-app** script. For example, browse to **software_examples/app,** highlight the **cpu2_led** folder, and click **OK.**
5. For **Project name,** enter your project name, for example **cpu2_led,** and click **Finish.**

If a dialog box appears with the message **Do you want the Nios II Software Build Tools for Eclipse to manage your makefile for you?,** click **Yes.**

6. Repeat steps 1 to 5 to import the remaining three projects.

Building the Software Projects

In this section, you build the software projects you just imported so they can be run on the processors in the system.

To build the software projects, perform the following steps:

1. In the Nios II perspective, right-click the project **cpu2_led** and click **Build Project**.
2. Right-click the project **niosII_multicore_socket_server** and click **Build Project.**

Creating a Debug Configuration for Each Processor

In this section, you create a run/debug configuration for each of the target processors. These configurations enable you to run and debug the two software projects you just built on the processors in the system.

To create a debug configuration for each processor, perform the following steps:

1. In the **Nios II** perspective, click the **cpu2_led** project.
2. On the Run menu, click **Debug Configurations**
3. In the configurations list, right-click **Nios II Hardware**.
4. Click **New**. A new debug configuration is created for the project.
5. Click the **Target Connection** tab.
6. If the **Name** column is not populated, click **Refresh Connections.**
7. If the **Name** column remains unpopulated, perform the following steps:
   a. In the **Project** tab, click **Advanced**. The **Nios II ELF Section Properties** dialog box appears.
   b. Under **Other**, set **JTAG Debugging Information File name** to <*working directory*>**/neek_vic_multi.jdi**.
   c. Click **Close.**
8. Under **Processors**, ensure that the row with **Name** value **cpu2** is selected.
9. Click **Apply**.
10. Click **Debug**. The Nios II SBT for Eclipse downloads and launches the **cpu2_led** software project on the **cpu2** processor, then pauses **cpu2** at a breakpoint set on `main()`.
11. If the **Debug Configurations** dialog box does not close automatically, click **Close** to return to the Nios II perspective.
12. If you are prompted to enter the Nios II Debug perspective, click Yes.
13. If you are not already in the Nios II Debug perspective, change to the Nios II Debug perspective by clicking the Debug perspective icon in the top right corner of your Nios II SBT for Eclipse window.
14. Check that the **cpu2** debug session, including the call stack, appears in the Nios II Debug perspective.

15. To return to the Nios II perspective, click the Nios II perspective icon in the top right corner of your Nios II SBT for Eclipse window. If the Nios II perspective icon is not visible, click the yellow plus-sign Open Perspective button.
16. Repeat steps 1–13 to create and run a debug configuration for the **cpu1** target processor, substituting **niosII_multicore_socket_server** for **cpu2_led** and **cpu1** for **cpu2**.

You have created, downloaded, and started a debug configuration for each processor in the system. You can now resume the code execution and debug code on each of the processors individually, using the normal flow for running or debugging.

Each processor begins executing code immediately after its code is downloaded to the FPGA; the processors do not start in unison. Although each processor begins running the code as soon as it is downloaded, the debug configuration ensures that the processor stops at a breakpoint set on `main()`.

Debugging the Software Projects on the Board

After you download both debug configurations to the NEEK, you must resume code execution on each processor.

To run the design example on the NEEK, after the two debug configurations are downloaded, perform the following steps:

1. To continue the **cpu2** debug run past the initial breakpoint, perform the following steps:
   a. To observe stepping in the debugger, click the `main()` call stack entry under the **cpu2** debug session.
   b. Click the Step Over icon in the toolbar menu to see **cpu2** step through the software code.
   c. To let **cpu2** run freely, click the green arrow Resume icon in the toolbar menu.
2. To continue the **cpu1** debug run past the initial breakpoint, perform the following steps:
   a. Click the `main()` call stack entry under the **cpu1** debug session.
   b. Click the green arrow Resume icon in the toolbar menu. The software running on the **cpu1** processor establishes an Ethernet link and displays an IP address for **cpu1** in the **Nios II Console** tab.
3. From the command line, telnet to **cpu1** at port 30 using the IP address displayed in the **cpu1** startup messages. For example, if the IP address displayed is 137.57.235.39, type the following command:

   `telnet 137.57.235.39 30` ↵
4. The LED Frequency Changing menu appears.
5. Enter values 0 through 9, and watch the corresponding blink rate change for LED1 on the NEEK.
6. To quit the telnet session, type `q`.

Conclusion

In this tutorial, you constructed, built software projects for, and debugged software on your first Nios II multiprocessor system. You have also learned how to use the **Mailbox** component to share system resources between processors. Feel free to experiment with the system you have created and find interesting new ways of using multiple processors in an Altera FPGA.

Altera recommends saving this system to use as a starting point next time you wish to create a multiprocessor system.

Additional Information

Revision History

The following table shows the revision history for this tutorial.

Date and Document Version	Changes Made	Summary of Changes
February 2010 v9.1 SP1	Updated for Nios II Software Build Tools for Eclipse.	—
December 2007 v1.3	Updated for Quartus II 7.2 release: minor text changes.	—
May 2007 v1.2	Updated for Quartus II 7.1 release.	—
May 2006 v1.1	Updated for Quartus II 6.0 release.	—
April 2005 v1.0	Initial release.	—

Referenced Documents

This tutorial references the following documents:

- *AN458: Alternative Nios II Boot Methods*
- *Mailbox Core* chapter in Volume 5: Embedded Peripherals of the *Quartus II Handbook*
- *MegaCore IP Library Release Notes and Errata*
- *Mutex Core* chapter in Volume 5: Embedded Peripherals of the *Quartus II Handbook*
- *Nios II Embedded Design Suite Release Notes and Errata*
- *Nios II Flash Programmer User Guide*
- *Nios II Hardware Development Tutorial*
- *Nios II Software Developer's Handbook*

How to Contact Altera

For the most up-to-date information about Altera products, refer to the following table.

Contact *(1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com

Contact (1)	Contact Method	Address
Documentation	Website	www.altera.com/literature
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

This document uses the typographic conventions shown in the following table.

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.
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.

© February 2010 Altera Corporation

Visual Cue	Meaning
↵	The angled arrow indicates you should press the Enter key.
(feet icon)	The feet direct you to more information about a particular topic.