ED51003-1.3

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 3–10

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 3–9

■ "Lauterbach Trace32 Debugger and PowerTrace Hardware" on page 3–10

■ "Insight and Data Display Debuggers" on page 3–10

Nios II Software Development Tools

The Nios II Software Build Tools for Eclipse™ is a graphical user interface (GUI) that supports creating, modifying, building, running, and debugging Nios II programs. The Nios II Software Build Tools for the command line are command-line utilities available from a Nios II Command Shell. The Nios II Software Build Tools for Eclipse use the same underlying utilities and scripts as the Nios II Software Build Tools for the command line. Using the Software Build Tools provides fine control over the build process and project settings.

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.

The following sections describe debugging tools and support features available in the Nios II software development tools:

■ "Nios II System ID"

■ "Project Templates"

■ "Configuration Options" on page 3–3

■ "Nios II GDB Console and GDB Commands" on page 3–6

■ "Nios II Console View 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
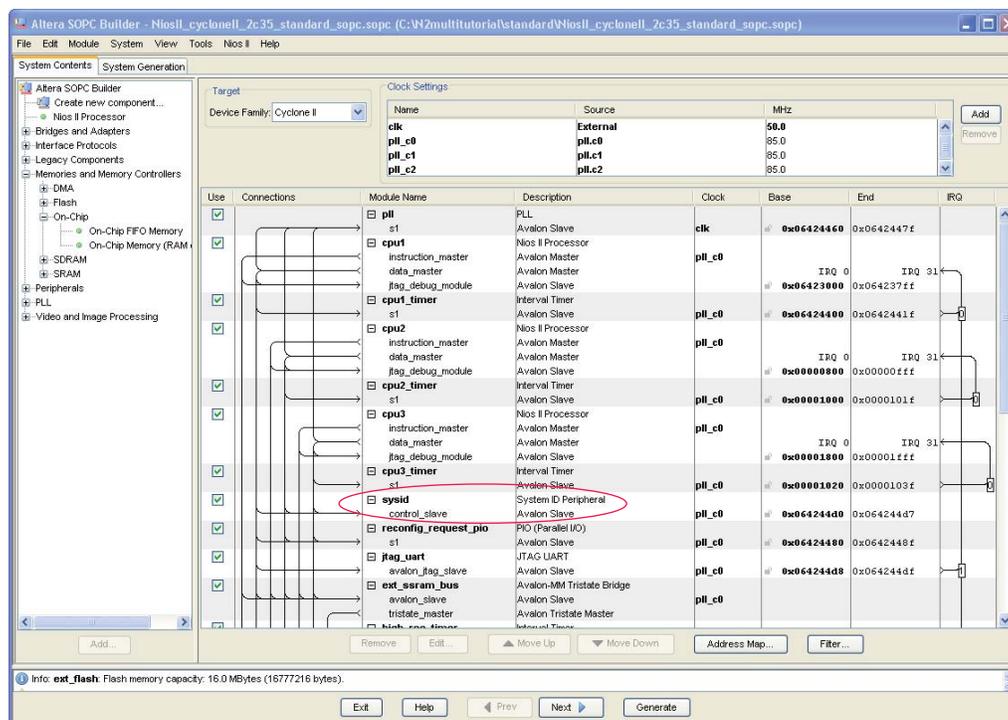
Nios II System ID

The system identifier (ID) feature is available as a system component in SOPC Builder.
The component allows the debugger to identify attempts to download software
projects with BSP projects that were generated for a different SOPC Builder system.
This feature protects you from inadvertently using an Executable and Linking Format
(**.elf)** file built for a Nios II hardware design that is not currently loaded in the FPGA.
If your application image does not run on the hardware implementation for which it
was compiled, the results are unpredictable.

To start your design with this basic safety net, in the Nios II Software Build Tools for
Eclipse **Debug Configurations** dialog box, in the **Target Connection** tab, ensure that
Ignore mismatched system ID is not turned on.

The system ID feature requires that the 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 Software Build Tools for Eclipse help you to create a simple, small, and
pretested software project to test a new board.

The Nios II Software Build Tools for Eclipse provide a mechanism to create new software projects using project templates. To create a simple test program to test a new board, perform the following steps:

1. In the Nios II perspective, on the File menu, point to **New**, and click **Nios II Application and BSP from Template**.

    The New Project wizard for Nios II C/C++ application projects appears.

2. Specify the SOPC information (**.sopcinfo**) file for your design. The folder in which this file is located is your project directory.

3. If your hardware design contains multiple Nios II processors, in the **CPU** list, click the processor you wish to run this application software.

4. Specify a project name.

5. In the **Templates** list, click **Hello World Small**.

6. Click **Next**.

7. Click **Finish**.

The Hello World Small template is a very simple, small application. Using a simple, small application minimizes the number of potential failures that can occur as you bring up a new piece of hardware.

To create a new project for which you already have source code, perform the preceding steps with the following exceptions:

■ In step 5, click **Blank Project**.

■ After you perform step 7, perform the following steps:

    a. Create the new directory *<your_project_directory>*/**software/***<project_name>*/**source**, where *<project_name>* is the project name you specified in step 4.

    b. Copy your source code files to the new project by copying them to the new *<your_project_directory>*/**software/***<project_name>*/**source** directory.

    c. In the **Project Explorer** tab, right-click your application project name, and click **Refresh**. The new **source** folder appears under your application project name.

Configuration Options

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

■ 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 Software Build Tools for Eclipse, perform the
following steps:

1. In the Project Explorer window, right-click your application project and click
    Properties.

2. On the list to the left, click **Nios II Application Properties**.

3. On the **Nios II Application Properties** page, turn on **Create object dump**.

4. Click **Apply**.

5. Click **OK**.

    After the next build, the **.objdump** file is found in the same directory as the
    newly built **.elf** file.

After the next build generates the **.elf** file, the build runs the **nios2-elf-objdump** command with the options `--disassemble-all`, `--source`, and `--all-headers` on the generated **.elf** file.

In the Nios II user-managed tool flow, you can edit the settings in the application makefile that determine the options with which the **nios2-elf-objdump** command runs. Running the **create-this-app** script, or the **nios2-app-generate-makefile** script, creates the following lines in the application makefile:

```
#Options to control objdump.
CREATE_OBJDUMP := 1
OBJDUMP_INCLUDE_SOURCE :=0
OBJDUMP_FULL_CONTENTS := 0
```

Edit these options to control the **.objdump** file according to your preferences for the project:

■ `CREATE_OBJDUMP`—The value 1 directs **nios2-elf-objdump** to run with the options `--disassemble`, `--syms`, `--all-header`, and `--source`.

■ `OBJDUMP_INCLUDE_SOURCE`—The value 1 adds the option `--source` to the **nios2-elf-objdump** command line.

■ `OBJDUMP_FULL_CONTENTS`—The value 1 adds the option `--full-contents` to the **nios2-elf-objdump** command line.

☞ For detailed information about the information each command-line option generates, in a Nios II Command Shell, type the following command:
`nios2-elf-objdump --help` ↵

Show Make Commands

To enable a verbose mode for the **make** command, in which the individual Makefile commands appear in the display as they are run, in the Nios II Software Build Tools for Eclipse, perform the following steps:

1. In the Project Explorer window, right-click your application project and click **Properties**.

2. On the list to the left, click **C/C++ Build**.

3. On the **C/C++ Build** page, turn off **Use default build command**.

4. For **Build command**, type `make -d`.

5. Click **Apply**.

6. Click **OK**.

Show Line Numbers

To enable display of C source-code line numbers in the Nios II Software Build Tools for Eclipse, follow these steps:

1. On the Window menu, click **Preferences**.

2. On the list to the left, under **General**, under **Editors**, select **Text Editors**.

3. On the **Text Editors** page, turn on **Show line numbers**.

4. Click **Apply**.

5. Click **OK**.

Nios II GDB Console and GDB Commands

The Nios II GNU Debugger (GDB) console allows you to send GDB commands to the Nios II processor directly.

To display this console, which allows you to enter your own GDB commands, click the blue monitor icon on the lower right corner of the Nios II Debug perspective. (If the Nios II Debug perspective is not displayed, on the Window menu, click **Open Perspective**, and click **Other** to view the available perspectives). If multiple consoles are connected, click the black arrow next to the blue monitor icon to list the available consoles. On the list, select the GDB console. 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

Console list icon
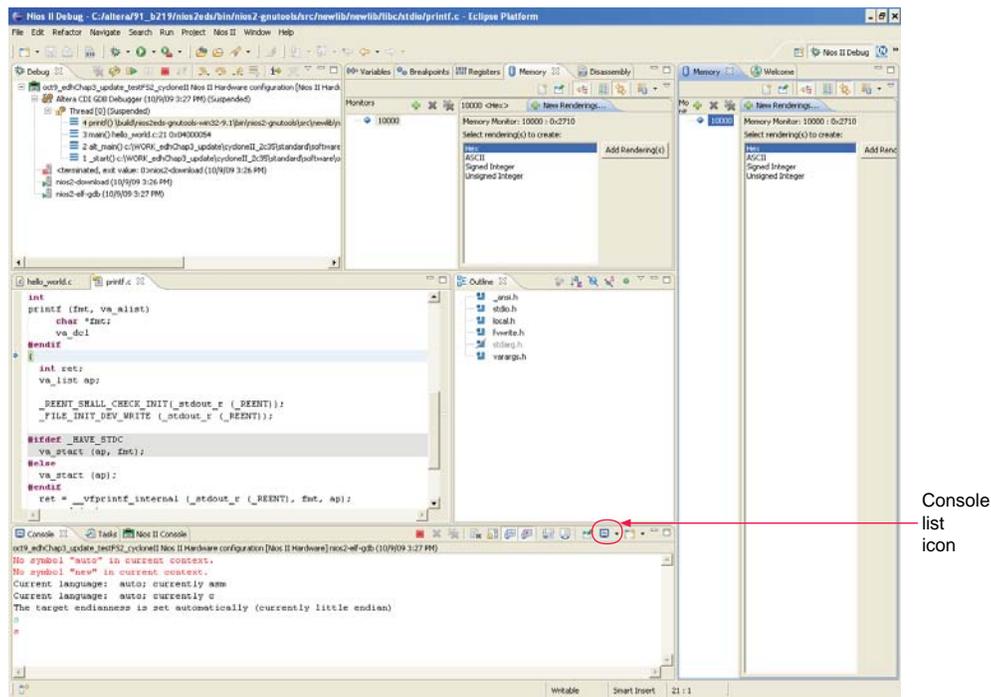
An example of a useful command you can enter in the GDB console is
```
dump binary memory <file> <start_addr> <end_addr> ↵
```

This command dumps the contents of a specified address range in memory to a file on the host computer. The file type is binary. You can view the generated binary file using the HexEdit hexadecimal-format editor that is available from the HexEdit website (www.expertcomsoft.com).

Nios II Console View and stdio Library Functions

When debugging I/O behavior, you should be aware of whether your Nios II software application outputs characters using the `printf()` function from the stdio library or the `alt_log_printf()` function. The two functions behave slightly differently, resulting in different system and I/O blocking behavior.

The `alt_log_printf()` function bypasses HAL device drivers and writes directly to the component device registers. The behavior of the two functions may also differ depending on whether you enable the reduced-driver option, whether you set your nios2-terminal session or the Nios II Console view in the Nios II Software Build Tools for Eclipse to use a UART or a jtag_uart as the standard output device, and whether the `O_NONBLOCK` control code is set. In general, enabling the reduced-driver option disables interrupts, which can affect blocking in jtag_uart devices.

To enable the reduced-drivers option, perform the following steps:

1. In the Nios II Software build Tools for Eclipse, in the Project Explorer window, right-click your BSP project.

2. Point to **Nios II** and click **BSP Editor**. The BSP Editor appears.

3. In the BSP Editor, in the **Settings** tab, under **Common**, under **hal**, click **enable_reduced_device_drivers**.

4. Click **Generate**.

For more information about the `alt_log_printf()` function, refer to "Using Character-Mode Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

### Importing Projects Created Using the Nios II Software Build Tools

Whether a project is created and built using the Nios II Software Build Tools for Eclipse or using the Nios II Software Build Tools command line, you can debug the resulting **.elf** image file in the Nios II Software Build Tools for Eclipse.

For information about how to import a project created with the Nios II Software Build Tools command line to the Nios II Software Build Tools for Eclipse, refer to "Importing a Command-Line Project" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II 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 an application project, the Nios II Software Build Tools for Eclipse require that you specify a Board Support Package (BSP) project. If a BSP for your application project does not yet exist, you can create one. For BSP generation, you must specify the CPU to which the application project is targeted.

Figure 3–3 shows how you specify the CPU for the BSP in the Nios II Software Build Tools for Eclipse. The **Nios II Board Support Package** page of the New Project wizard collects the information required for BSP creation. This page derives the list of available CPU choices from the **.sopcinfo** file for the system.

Figure 3–3. Nios II Software Build Tools for Eclipse Board Support Package Page— CPU Selection
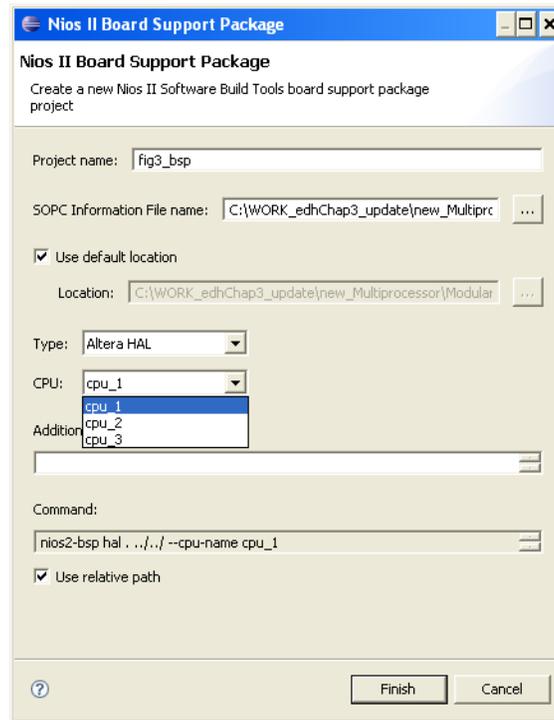
From the Nios II Command Shell, the **jtagconfig –n** command identifies available JTAG devices and the number of CPUs in the subsystem connected to each JTAG device. Example 3–2 shows the system response to a **jtagconfig -n** command.

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

```
[SOPC Builder]$ jtagconfig -n
1) USB-Blaster [USB-0]
 120930DD EP2S60
 Node 19104600
 Node 0C006E00
2) USB-Blaster [USB-1]
 020B40DD EP2C35
 Node 19104601
 Node 19104602
 Node 19104600
 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 0x191046xx in the response confirms that your FPGA implementation has a Nios II processor with a JTAG debug

module. The appearance of a node number `0x0C006Exx` in the response confirms that the FPGA implementation has a JTAG UART component. The CPU instances are identified by the least significant byte of the nodes beginning with `191`. The JTAG UART instances are identified by the least significant byte of the nodes beginning with `0C`. Instance IDs begin with `0`.

Only the CPUs that have JTAG debug modules appear in the listing. Use this listing to confirm you have created JTAG debug modules for the Nios II processors you intended.

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 using the Nios II Software Build Tools command line, 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 <*Nios II EDS install path*>**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 Software Build Tools for Eclipse 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 and an instruction-set simulator for Nios II systems.

The order in which devices are powered up is critical. The Lauterbach PowerTrace hardware must always be powered when power to the FPGA hardware is applied or terminated. The Lauterbach PowerTrace hardware's protection circuitry is enabled after the module is powered up.

For more information about the Lauterbach PowerTrace hardware and the required power-up sequence, refer to *AN543: Debugging Nios II Software Using the Lauterbach Debugger*.

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 Software Build Tools for Eclipse, 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 Software Build Tools for Eclipse support watchpoints directly.

For more information about watchpoints, refer to the Nios II online Help. In Nios II Software Build Tools for Eclipse, on the Help menu, click **Search**. In the search field, type `watchpoint`, and select the topic **Adding watchpoints**.

To enable watchpoints, you must configure the Nios II processor's debug level in 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, right-click the desired Nios II processor component. A list of options appears.

2. On the list, click **Edit**. The Nios II processor configuration page appears.

3. Click the **JTAG Debug Module** tab, shown in Figure 3–4 on page 3–13.

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 3–4 on page 3–13 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

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.

Using the Nios II Software Build Tools for Eclipse, you can enable the HAL to check for stack overflow. If you enable stack overflow checking and you register an instruction-related exception handler, on stack overflow, the HAL calls the instruction-related exception handler. If you enable stack overflow checking and do not register an instruction-related exception handler, and you enable a JTAG debug module for your Nios II processor, on stack overflow, execution pauses in the debugger, exactly as it does when the debugger encounters a breakpoint. To enable stack overflow checking, in the BSP Editor, in the **Settings** tab, under **Advanced**, under **hal**, click **enable_runtime_stack_checking**.

For information about the instruction-related exception handler, refer to "The Instruction-Related Exception Handler" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

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, right-click the desired Nios II processor component.

2. On the right button pop-up menu, click **Edit**. The Nios II processor configuration page appears.

3. Click the **JTAG Debug Module** tab, shown in Figure 3–4.

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 3–4 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 3–4. Nios II Processor — JTAG Debug Module — SOPC Builder Configuration Page
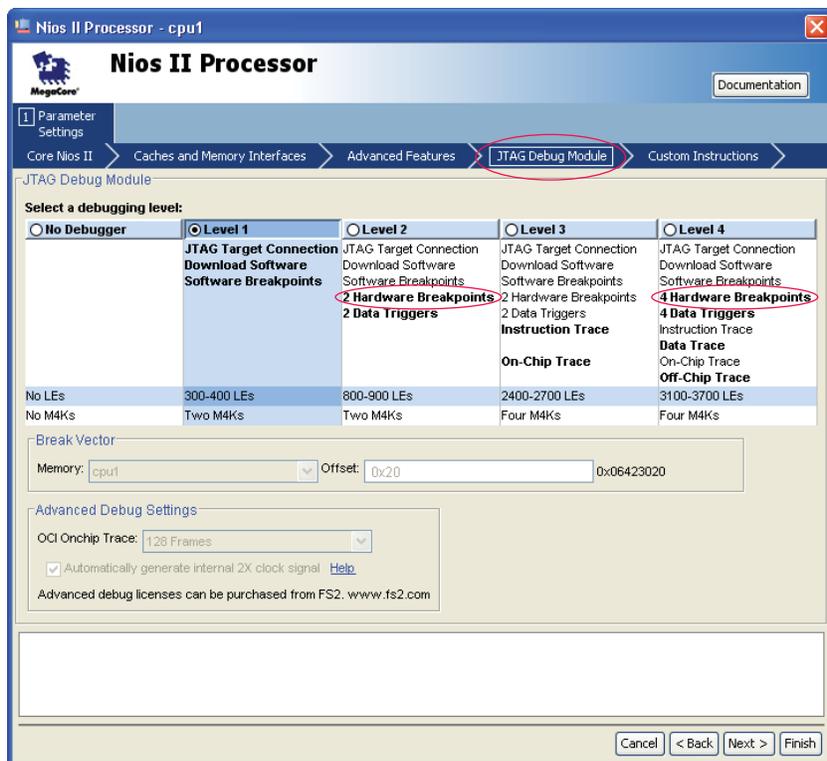
For more information about the Nios II processor debug levels, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

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 Software Build Tools for Eclipse, perform the following steps:

1. In the Nios II perspective, right-click your application project. A list of options appears.

2. On the list, click **Properties**. The **Properties for** *<project name>* dialog box appears.

3. In the left pane, click **Nios II Application Properties**.

4. In the **Optimization Level** list, select **Off**.

5. Click **Apply**.

6. Click **OK**.

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*

■ *AN543: Debugging Nios II Software Using the Lauterbach Debugger*

■ *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*

■ *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*

■ *Exception Handling* chapter of the *Nios II Software Developer's Handbook*

■ *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*

■ *Getting Started with the Graphical User Interface* 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
December 2009 v1.3	■ Updated for Nios II Software Build Tools for Eclipse. ■ Removed all Nios II IDE instructions. Replaced all instances of Nios II IDE instructions with instructions for Nios II Software Build Tools for Eclipse.	Updated for Nios II Software Build Tools for Eclipse.
April 2009 v1.2	■ Added reference to new application note *AN543: Debugging Nios II Software Using the Lauterbach Debugger* ■ Removed information made redundant by the new application note from "Lauterbach Trace32 Debugger and PowerTrace Hardware" section.	Removed information about the Lauterbach debugging tools now described in new application note.
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release.	—