Profiling Nios II Systems

© May 2010 AN-391-2.0

This application note describes a variety of ways to measure the performance of a Nios® II system with three tools: the GNU profiler, called **nios2-elf-gprof**, the timestamp interval timer component, and the performance counter component. Two tutorials give detailed examples of using these tools to measure performance in the Altera® Nios II Software Build Tools (SBT) development flow.

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

The performance counter component and the timestamp component are minimally intrusive hardware methods for measuring the performance of a Nios II system. The application note describes and compares the two components. To use these methods, you add the hardware components to your system, and you add macro invocations to your source code to start and stop the components. The hardware components perform accurate measurements.

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

Requirements

The tutorials assume you are familiar with the Nios II SBT development flow for Nios II systems, including use of the Quartus® II software and SOPC Builder.

Obtaining the Hardware Design

The tutorials in this application note work with the standard or Board Update Portal (BUP) hardware design for any Altera development kit. You can find the standard or BUP hardware design for your development kit through the All Development Kits page of the Altera website.

For example, you can find the standard Verilog HDL hardware design for the Nios II Embedded Evaluation Kit (NEEK), Cyclone® III Edition through the Nios II Embedded Evaluation Kit (NEEK), Cyclone III Edition page of the Altera website.

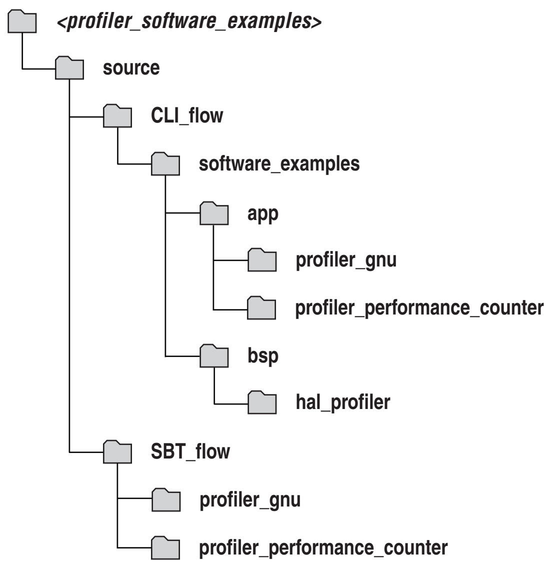
To use the design, unzip the files to a working directory on your system. For convenience, this application note refers to that directory as <*project_directory*>.

Obtaining the Software Examples

To obtain the software examples for this application note, perform the following steps:

1. Download the file **profiler_software_examples.zip**. You can find this file on the Literature: Nios II Processor page of the Altera website, with this application note.
2. Unzip the file **profiler_software_examples.zip** to a directory on your system. For convenience, the application note refers to that directory as <*profiler_software_examples*>. The directory structure shown in Figure 1 appears.

Figure 1. Directory Structure After Unzipping Files


<profiler_software_examples>
source
CLI_flow
software_examples
app
profiler_gnu
profiler_performance_counter
bsp
hal_profiler
SBT_flow
profiler_gnu
profiler_performance_counter

Finding Nios II EDS Files

When you install the Nios II EDS, you specify a root directory for the EDS file structure. For example, if the Nios II EDS 9.1 is installed on the Windows operating system, the root directory might be **c:/altera/91/nios2eds**.

For simplicity, this document refers to the directory as <*Nios II EDS install path*>.

Tools

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

- The GNU Profiler
- The Altera Performance Counter
- The High-Resolution Timer

In addition, the program counter trace collection tool is available for some Nios II processors. This tool is not used in the tutorials.

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

The GNU Profiler

Minimal source code changes are required to take measurements for analysis with the GNU profiler. To implement the required changes, perform the following steps:

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

   If you are using the Nios II SBT for Eclipse, `hal.enable_exit` is on by default.

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

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

3. Rebuild the BSP and the application project.

The Altera Performance Counter

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

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

You can change the maximum number of measured code sections by editing the performance counter component in SOPC Builder.

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

Performance counters are best suited for analyzing determinism and other run-time issues.

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

The High-Resolution Timer

A high-resolution timer, in contrast to a performance counter component, does not use a large number of LEs on your FPGA, and does not require heavy instrumentation of every function call in your code to obtain performance measurements. Timers require explicit calls to read the timer in the sections of the source code that you want to measure, so their use is better suited for pinpointing the performance issues in a program. You instrument the source code manually, but because this instrumentation is less pervasive, it is also less intrusive. Many more processor cycles are required to make two function calls—one to read the time at the beginning of a measured section, and one to read the time at the end—than are consumed by the performance counter macros.

The Program Counter Trace Information

The Nios II processor can generate complete and accurate program counter trace information. This information is not used by the GNU profiler. To generate this information you must have a Nios II processor configured with a JTAG debug module of level 3 or greater. The level 3 JTAG debug module creates on-chip trace data. Approximately a dozen instructions can be captured in the on-chip trace buffer. You can obtain a much larger trace by configuring a Nios II core with a level 4 JTAG debug module to generate off-chip trace information. Collecting this off-chip trace data requires the First Silicon Solutions, Inc. (FS2) or Lauterbach Datentechnik GmBH (Lauterbach) (www.lauterbach.com) hardware.

For more information about the Lauterbach hardware, refer to “Debuggers” in the *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*.

Using the GNU Profiler to Measure Code Performance

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

GNU Profiler Advantages

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

GNU Profiler Drawbacks

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

- Each function is slightly larger because of the additional function call to collect profiling information.
- Collecting the profiling information increases the entry and exit time of each function.

- Pulling the profiling function into instruction cache memory generates more instruction-cache misses than are generated by the original source code.
- Memory used to record the profiling data can change the behavior of the data cache.

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

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

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

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

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

Software Considerations

The profiler instruments your source code with functions to track processor utilization.

Profiler Mechanics

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

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

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

1. The compiler instruments function prologues with a call to `mcount()` to enable it to determine the function call graph. In the GNU profiler documentation, this data is called the function call arcs.
2. An alarm is registered with the timer interrupt handler to capture information about the foreground function that is executing when the alarm triggers (this data is called histogram data).
3. The profiling data is stored in target memory allocated from the heap.
4. When your code exits with a `BREAK 2` instruction, the **nios2-download** utility copies the profiling data from the target to the host.

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

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

Profiler Overhead

Using the profiler impacts both memory and processor cycles.

Memory

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

To view the impact on the `.text` section, you can compare the sizes of the `.text` sections in the **.objdump** file when profiling is enabled and when it is not enabled.

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

((`.text` section size) / 32) × 2 bytes

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

Processor Cycles

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

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

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

Hardware Considerations

The profiler requires only a system timer. No special components are required. If your Nios II hardware design already includes a system timer component, you do not need to change the design.

Tutorial: Using the GNU Profiler

For this tutorial, use the standard or BUP hardware design example for your Altera development board. If your development board is programmed with another hardware design, follow the next few steps to program the standard or BUP hardware design. If the development board already has the standard or BUP hardware design programmed, proceed to "Creating the Profiler Software Example".

To program the FPGA with the standard or BUP hardware design for your Altera development board, perform the following steps:

1. Start the Quartus II software, version 9.1 or later.
2. In the Quartus II window, on the New menu, click **Open Project**.
3. Open the Quartus II project file for the standard or BUP hardware design for your board.
4. On the Tools menu, click **Programmer**.
5. Click **Start** to download the Nios II SRAM Object File (**.sof**) to the FPGA.

☞ If the **Start** button is disabled, or the USB-Blaster™ cable is not listed in the **Hardware Setup** field, refer to the *Introduction to the Quartus II Software* manual for more details on the Programmer tool.

Profiler Example with the Nios II Command Line

Creating the Profiler Software Example

To create the profiler_gnu software project in the Nios II command-line flow, perform the following steps:

1. Open a Nios II Command Shell by executing one of the following steps, depending on your environment:
   - In the Windows operating system, on the Start menu, point to **Programs** > **Altera** > **Nios II EDS** <*version*>, and click **Nios II** <*version*> **Command Shell**.
   - In the Linux operating system, in a command shell, change directories to <*Nios II EDS install path*>, and type the command `./sdk_shell`.
2. Change to the working directory for your hardware and software projects (<*project_directory*>).
3. Ensure that the working directory <*project_directory*> and all of its subdirectories are write-enabled, by typing the following command:

   ```
   chmod -R +w . ↵
   ```
4. Change to the directory **software_examples/app/profiler_gnu**.
5. Create and build the application with the **create-this-app** script, by typing the following command:

   ```
   ./create-this-app ↵
   ```

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

```
set_setting hal.enable_gprof true
set_setting hal.enable_exit true
```

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

Running the Profiler Software Example

To run the application and collect the GNU profiler data, perform the following steps:

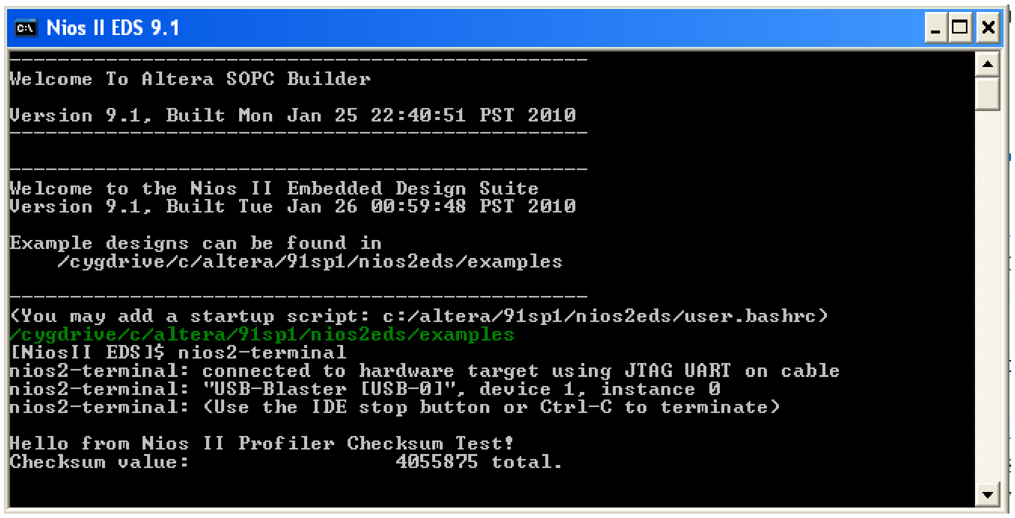
1. Open a second Nios II Command Shell.
2. In the second shell, open a **nios2-terminal** session by typing the following command:

   ```
   nios2-terminal ↵
   ```
3. In your original Nios II Command Shell, download the **.elf** file to the development board, run the design, and write the GNU profiler data to the **gmon.out** file, by typing the following command:

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

   The GNU profiler collects data while the application runs, and then writes the data to the **gmon.ou**t file when the application calls the `exit()` function. Figure 2 shows an example of the output in the Nios II Command Shell.
4. Exit **nios2-terminal** by typing control-C.

Figure 2. GNU Profiler Output on nios2-terminal



```
------------------------------------------------
Welcome To Altera SOPC Builder

Version 9.1, Built Mon Jan 25 22:40:51 PST 2010
------------------------------------------------

------------------------------------------------
Welcome to the Nios II Embedded Design Suite
Version 9.1, Built Tue Jan 26 00:59:48 PST 2010

Example designs can be found in
    /cygdrive/c/altera/91sp1/nios2eds/examples

------------------------------------------------
(You may add a startup script: c:/altera/91sp1/nios2eds/user.bashrc)
/cygdrive/c/altera/91sp1/nios2eds/examples
[NiosII EDS]$ nios2-terminal
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Hello from Nios II Profiler Checksum Test!
Checksum value:                  4055875 total.
```

Creating the Profiler Report

When you run the project, it creates the **gmon.out** file. Format this file in a readable format by performing the following steps:

1. In the original Nios II Command Shell, change directory to *<project_directory>***/software_examples/app/profiler_gnu**.
2. Type the following command:

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

   This command generates a flat profile report and a call graph, which are captured in the file **report.txt**.
3. Use any text editor to view the **report.txt** file.

Refer to "Analyzing the GNU Profiler Report" on page 11 for help understanding the information in the report.

Profiler Example with Nios II SBT for Eclipse

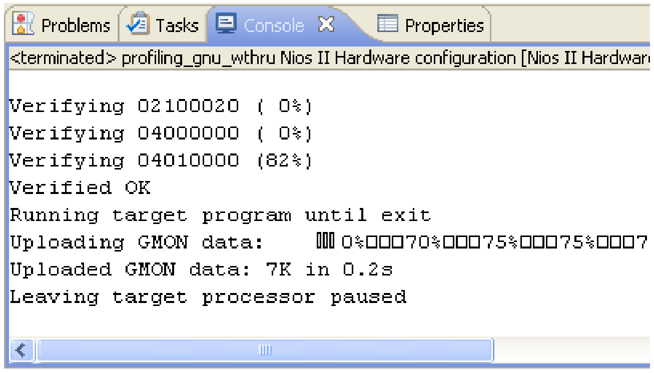
Creating and Running the Profiler Software Example

1. Start the Nios II SBT for Eclipse.
2. Right-click in the Project Explorer view, point to **New**, and click **Nios II Application and BSP from template**.
3. Set **SOPC Information File name** by browsing to locate the SOPC Information File (**.sopcinfo**) in *<project_directory>*.
4. Give your project a name, such as profiler_gnu.
5. Under **Templates**, select **Blank Project**.
6. Click **Finish** to create your new project.

7. Locate the **profiler_software_examples/source/SBT_flow/profiler_gnu** folder. Add all the files in this directory to your application project folder. For example, drag and drop the files from a file management tool to the application project folder in the Project Explorer view in the Nios II SBT for Eclipse.
8. Right-click your project in the Project Explorer view, point to **Nios II** and click **BSP Editor**.
9. In the Nios II BSP Editor, turn on `hal.enable_gprof` to enable the GNU profiler in your project.
10. Save and generate your BSP project.
11. Exit the BSP Editor.
12. Build your project in Nios II SBT for Eclipse.
13. To download and run the profiler_gnu software, right-click on your application project, point to **Run As** and click **Nios II Hardware**.

Figure 3 illustrates the output that appears on the Eclipse console after the software is downloaded.

Figure 3. Eclipse Console After Downloading Program Into Board

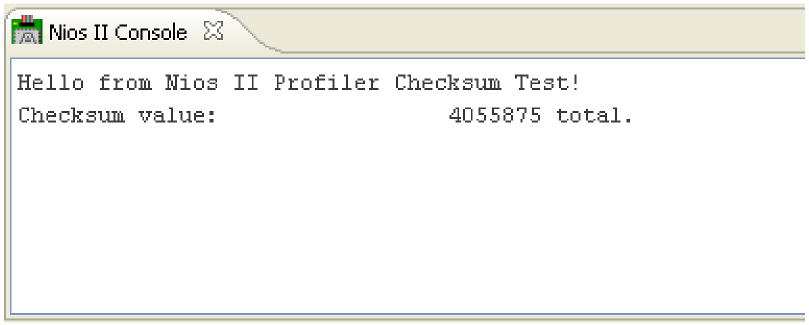


```
<terminated> profiling_gnu_wthru Nios II Hardware configuration [Nios II Hardwar

Verifying 02100020 ( 0%)
Verifying 04000000 ( 0%)
Verifying 04010000 (82%)
Verified OK
Running target program until exit
Uploading GMON data:    ▯▯▯ 0%▯▯▯70%▯▯▯75%▯▯▯75%▯▯▯7
Uploaded GMON data: 7K in 0.2s
Leaving target processor paused
```

Figure 4 illustrates the software's output to the Nios II console.

Figure 4. Nios II Console After Running profiler_gnu



```
Hello from Nios II Profiler Checksum Test!
Checksum value:                  4055875 total.
```

Viewing the Profiler Report

After the software exits, the **gmon.out** file is created in your project folder, and appears in the Project Explorer view of the Nios II SBT for Eclipse. When you open **gmon.out**, the Nios II SBT for Eclipse switches to the Profiling perspective, where you can view the report. Refer to "Analyzing the GNU Profiler Report" for help in understanding the information in the report.

Analyzing the GNU Profiler Report

The information in this section is applicable to the GNU profiler report regardless whether it is generated from the command line or through the Nios II SBT for Eclipse.

The profiler report contains information in the following two formats:

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

Refer to the Nios II GNU profiler documentation, included with the GCC documentation, available at the Nios II Embedded Design Suite Support page of the Altera website.

The profiler report excerpts shown in Example 1 were generated by running the GNU profiler tutorial software on a NEEK, containing a Cyclone III 3c25 device with the Nios II version 9.1 standard hardware design running at 100 MHz.

In Example 1, the call graph shows that the `checksum_test_routine()` function call (index[4]) consumed 86.9% of the processing time during the execution.

The granularity statement in the call graph report states that the report covers 3.96 seconds (3960 milliseconds). The Nios II timer (**sys_clk_timer**) has a 10 millisecond timer. The timer interrupt is called once at the beginning, before a full clock period has elapsed, and once every 10 milliseconds thereafter. A precise report, therefore, would show that the timer interrupt handler is called 396 times. Index[22] shows that `alt_avalon_timer_sc_irq()` is called 399 times, which is within the precision range of this measurement method.

Example 1. Flat Profile and Call Graph Example

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 85.86      3.40     3.40        1     3.40     3.44  checksum_test_routine
 10.83      3.83     0.43        1     0.43     0.43  alt_busy_sleep

        .
        .
        .

           Call graph (explanation follows)


granularity: each sample hit covers 32 byte(s) for 0.25% of 3.96 seconds

index      % time    self  children    called     name
                3.40    0.04       1/1           main [2]
[4]            86.9    3.40    0.04       1         checksum_test_routine [4]
                0.04    0.00   32000/32052       udivmodsi4 [8]
                0.00    0.00   32000/32024       __umodsi3 [129]
                0.00    0.00     300/300         alt_dcache_flush_all [19]

*************
                0.00    0.00     414/414         alt_exception [69]
[16]           0.0    0.00    0.00     414         alt_irq_handler [16]
                0.00    0.00     399/399         alt_avalon_timer_sc_irq [17]
                0.00    0.00      15/15          altera_avalon_jtag_uart_irq [21]

        .
        .
        .
```

Using Performance Counter and Timer Components

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

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

Performance Counter Advantages

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

Timer Advantages

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

Performance Counter and Timer Hardware Considerations

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

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

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

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

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

Performance Counter and Timer Software Considerations

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

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

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

Performance Counter Software Considerations

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

PERF_BEGIN and PERF_END are single writes to the performance counter component. These macros are very efficient, requiring only two or three machine instructions. They are defined in **altera_avalon_performance_counter.h** as follows:

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

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

The Global Counter

The performance counter component contains a number of counters. You can configure the number of measured sections in SOPC Builder. Normally, you have one pair of counters for each measured section, as described in "The Altera Performance Counter" on page 3. In addition, the performance counter component always has a global counter.

The global counter measures the total time during which measurements are being taken. None of the other counters are allowed to run when the global counter is stopped. Special macros—PERF_START_MEASURING and PERF_STOP_MEASURING—control the global counter. Do not attempt to manipulate the global counter in any other way.

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

Hardware Considerations

Performance counters and timestamp interval timers are SOPC Builder components. When you add one to an existing system, you must regenerate the SOPC Builder system and recompile the .**sof** file in the Quartus II software. Timers and performance counters can eventually overflow, like any hardware counter.

Tutorial: Using Performance Counters and Timers

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

In this tutorial, you use the same NEEK design as the GNU profiler tutorial. This design has an interval timer and a performance counter. You change the timer interval and the number of sections measured by the performance counter.

Refer to "Requirements" on page 1 for instructions for downloading the design file.

Creating the Performance Counter Hardware Design

To create the standard_perf_counter hardware design, perform the following steps:

1. Make a copy of <*project_directory*> (the design used in "Tutorial: Using the GNU Profiler" on page 7), and rename it to **standard_perf_counter**.
2. Start the Quartus II software, version 9.1 or later.
3. In the Quartus II window, on the New menu, click **Open Project**.
4. Open the Quartus II project file for the hardware design you just copied. For example, the standard project file for the NEEK is the file **cycloneIII_3c25_niosII_standard.qpf**, located in your new directory **standard_perf_counter**.
5. On the Tools menu, click **SOPC Builder**. SOPC Builder starts.
6. Right-click the **performance_counter** module and click **Edit**.
7. Set **Number of simultaneously-measured sections** to 3.
8. Click **Finish**.

   When you increase the number of sections in the performance counter, errors such as the following might appear in the Quartus II console:

```
Error: slow_peripheral_bridge.m1: performance_counter.control_slave \
    cannot be at 0x120 (0x100 or 0x140 are acceptable)
Error: slow_peripheral_bridge.m1: pll.s1 (0x140..0x15f) overlaps \
    performance_counter.control_slave (0x120..0x15f)
```

   In this case, correct the module base addressing as follows: in the System menu, click **Auto-Assign Base Addresses**.

   If your design does not have a **performance_counter** module, add one. Set up **performance_counter** as a Performance Counter Unit module with three sections.

9. Right-click the **high_res_timer** module and click **Edit**.
10. Under **Timeout period**, set the interval time's **Period** to 1 and the units to **us** (microseconds).
11. Click **Finish**.

    If your design does not have a **high_res_timer** module, add one. Set up **high_res_timer** as an Interval Timer with a one-microsecond period, a 32-bit counter and Full-featured presets.

    Figure 5 shows the SOPC Builder system after you have made these modifications.

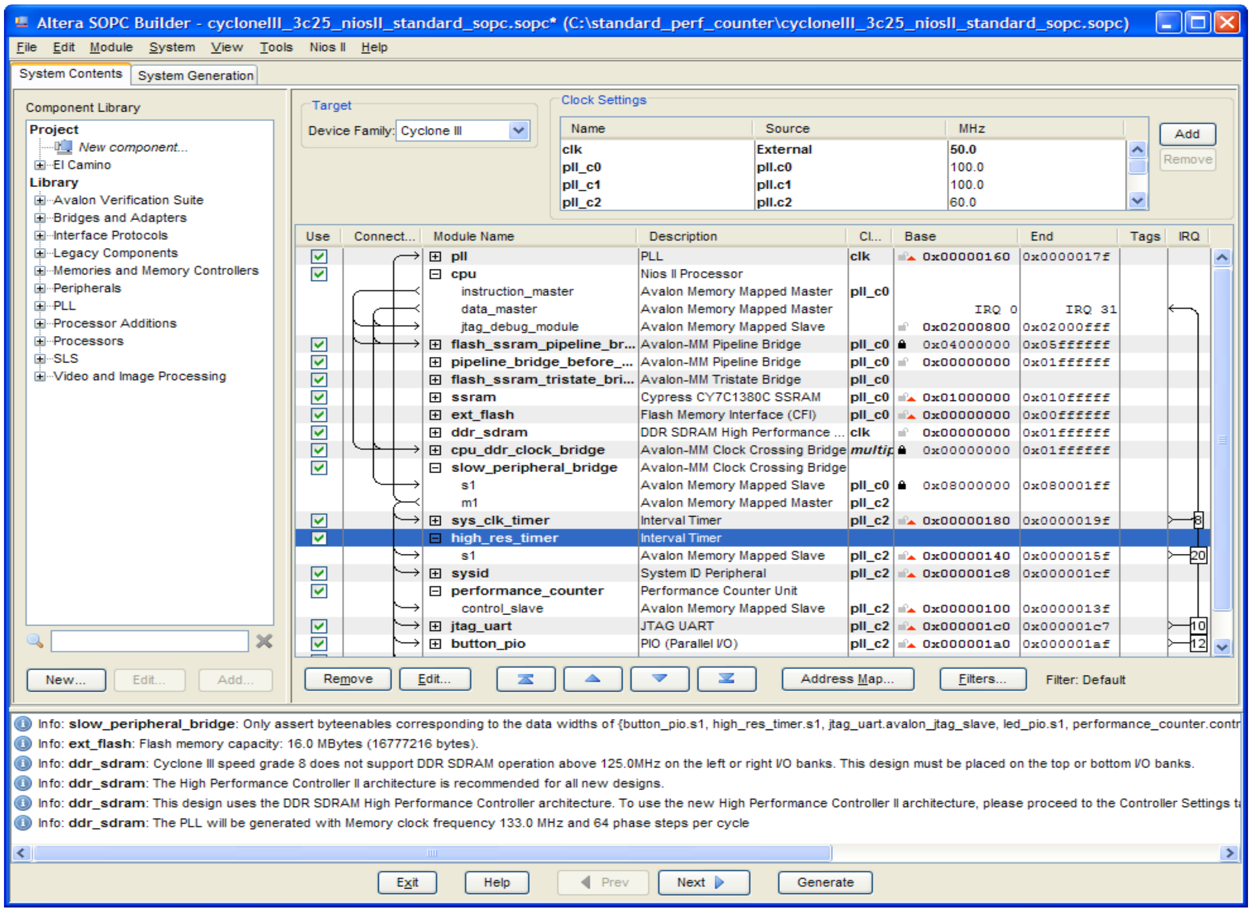
12. On the File menu, click **Save**.
13. Click the **System Generation** tab.
14. Click **Generate**. The generation phase takes a few minutes.

15. When system generation is complete, the following message appears:

    ```
    SUCCESS: SYSTEM GENERATION COMPLETED
    ```

    Click **Exit**. The hardware design is ready to be compiled by the Quartus II software.

Figure 5. SOPC Builder Window


Altera SOPC Builder - cycloneIII_3c25_niosII_standard_sopc.sopc* (C:\standard_perf_counter\cycloneIII_3c25_niosII_standard_sopc.sopc)
File Edit Module System View Tools Nios II Help
System Contents
System Generation
Component Library
Project
New component...
El Camino
Library
Avalon Verification Suite
Bridges and Adapters
Interface Protocols
Legacy Components
Memories and Memory Controllers
Peripherals
PLL
Processor Additions
Processors
SLS
Video and Image Processing
New... Edit... Add...
Target
Clock Settings
Add
Remove
Remove Edit... Address Map... Filters...
Exit Help Prev Next Generate

Device Family: Cyclone III

Name	Source	MHz
clk	External	50.0
pll_c0	pll.c0	100.0
pll_c1	pll.c1	100.0
pll_c2	pll.c2	60.0

Use	Connect...	Module Name	Description	Cl...	Base	End	Tags	IRQ
✓		pll	PLL	clk	0x00000160	0x0000017f		
✓		cpu	Nios II Processor					
		instruction_master	Avalon Memory Mapped Master	pll_c0				
		data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31		
		jtag_debug_module	Avalon Memory Mapped Slave		0x02000800	0x02000fff		
✓		flash_ssram_pipeline_br...	Avalon-MM Pipeline Bridge	pll_c0	0x04000000	0x05ffffff		
✓		pipeline_bridge_before_...	Avalon-MM Pipeline Bridge	pll_c0	0x00000000	0x01ffffff		
✓		flash_ssram_tristate_bri...	Avalon-MM Tristate Bridge	pll_c0				
✓		ssram	Cypress CY7C1380C SSRAM	pll_c0	0x01000000	0x010fffff		
✓		ext_flash	Flash Memory Interface (CFI)	pll_c0	0x00000000	0x00ffffff		
✓		ddr_sdram	DDR SDRAM High Performance ...	clk	0x00000000	0x01ffffff		
✓		cpu_ddr_clock_bridge	Avalon-MM Clock Crossing Bridge	*multip*	0x00000000	0x01ffffff		
✓		slow_peripheral_bridge	Avalon-MM Clock Crossing Bridge					
		s1	Avalon Memory Mapped Slave	pll_c0	0x08000000	0x080001ff		
		m1	Avalon Memory Mapped Master	pll_c2				
✓		sys_clk_timer	Interval Timer	pll_c2	0x00000180	0x0000019f		8
✓		high_res_timer	Interval Timer					
		s1	Avalon Memory Mapped Slave	pll_c2	0x00000140	0x0000015f		20
✓		sysid	System ID Peripheral	pll_c2	0x000001c8	0x000001cf		
✓		performance_counter	Performance Counter Unit					
		control_slave	Avalon Memory Mapped Slave	pll_c2	0x00000100	0x0000013f		
✓		jtag_uart	JTAG UART	pll_c2	0x000001c0	0x000001c7		10
✓		button_pio	PIO (Parallel I/O)	pll_c2	0x000001a0	0x000001af		12

Filter: Default

```
Info: slow_peripheral_bridge: Only assert byteenables corresponding to the data widths of {button_pio.s1, high_res_timer.s1, jtag_uart.avalon_jtag_slave, led_pio.s1, performance_counter.contr
Info: ext_flash: Flash memory capacity: 16.0 MBytes (16777216 bytes).
Info: ddr_sdram: Cyclone III speed grade 8 does not support DDR SDRAM operation above 125.0MHz on the left or right I/O banks. This design must be placed on the top or bottom I/O banks.
Info: ddr_sdram: The High Performance Controller II architecture is recommended for all new designs.
Info: ddr_sdram: This design uses the DDR SDRAM High Performance Controller architecture. To use the new High Performance Controller II architecture, please proceed to the Controller Settings t
Info: ddr_sdram: The PLL will be generated with Memory clock frequency 133.0 MHz and 64 phase steps per cycle
```

16. In the Processing menu in Quartus II, click **Start Compilation**. This step generates the **.sof** file.
17. Click **OK** when you see the following message:

    ```
    Full Compilation was successful
    ```

Programming the Hardware Design to an FPGA

After you have compiled your modified hardware design, you can program it to the FPGA by performing the following steps:

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

If the **Start** button is greyed out, or the USB-Blaster cable is not listed in the **Hardware Setup** field, refer to the *Introduction to the Quartus II Software* manual for more details on the Programmer tool.

Performance Counter Example with the Nios II Command Line

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

Creating the Performance Counter Software Example

To create the profiler_performance_counter software project in the Nios II software build flow, perform the following steps:

1. Open a Nios II Command Shell as described in "Creating the Profiler Software Example" on page 8.
2. Change to the **standard_perf_counter** directory.
3. Ensure that the directory **standard_perf_counter** and all of its subdirectories are write-enabled, by typing the following command:

   ```
   chmod -R +w . ↵
   ```

4. Change to the directory **software_examples/app/profiler_performance_counter**.
5. Create and build the application by typing the following command:

   ```
   ./create-this-app ↵
   ```

   The **create-this-app** script runs the **create-this-bsp** script, which reads settings from the **parameter_definition.tcl** file in **standard_perf_counter/ software_examples/bsp/hal_profiler**. This Tcl file contains the following two lines:

   ```
   set_setting hal.enable_gprof true
   set_setting hal.enable_exit true
   ```

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

6. Edit the **parameter_definition.tcl** file by adding the following two lines before the two lines described in Step 5:

   ```
   set_setting hal.sys_clk_timer sys_clk_timer
   set_setting hal.timestamp_timer high_res_timer
   ```

   The new lines configure the HAL BSP settings to use the appropriate SOPC Builder system components.

Running the Performance Counter Software Example

To run the application and collect the GNU profiler data, perform the following steps:

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

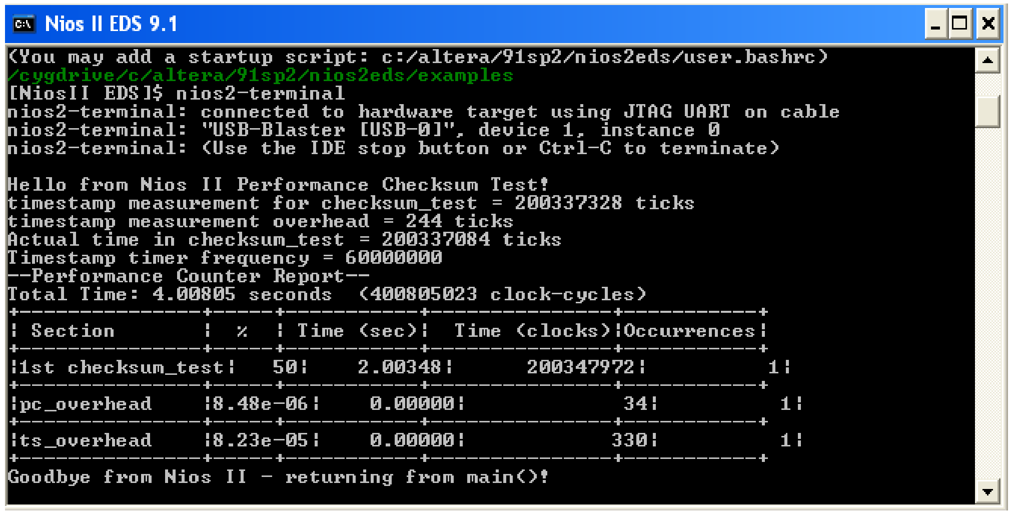
   ```
   nios2-terminal ↵
   ```

3. In your original Nios II Command Shell, write the **.elf** file to the development board, run the design, and write the performance data to **nios2-terminal**, by typing the following command:

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

Figure 6 shows an example of the output that appears in the Nios II Command Shell. Your output might vary. Refer to "Analyzing the Performance Counter Report" for help in understanding the information in the report.

Figure 6. Performance Counter Report on nios2-terminal



```
(You may add a startup script: c:/altera/91sp2/nios2eds/user.bashrc)
/cygdrive/c/altera/91sp2/nios2eds/examples
[NiosII EDS]$ nios2-terminal
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Hello from Nios II Performance Checksum Test!
timestamp measurement for checksum_test = 200337328 ticks
timestamp measurement overhead = 244 ticks
Actual time in checksum_test = 200337084 ticks
Timestamp timer frequency = 60000000
--Performance Counter Report--
Total Time: 4.00805 seconds  (400805023 clock-cycles)
+----------------+-----+-----------+---------------+-----------+
| Section        |  %  | Time (sec)|  Time (clocks)|Occurrences|
+----------------+-----+-----------+---------------+-----------+
|1st checksum_test|   50|    2.00348|      200347972|          1|
+----------------+-----+-----------+---------------+-----------+
|pc_overhead     |8.48e-06|    0.00000|             34|          1|
+----------------+-----+-----------+---------------+-----------+
|ts_overhead     |8.23e-05|    0.00000|            330|          1|
+----------------+-----+-----------+---------------+-----------+
Goodbye from Nios II - returning from main()!
```

Performance Counter Example with Nios II SBT for Eclipse

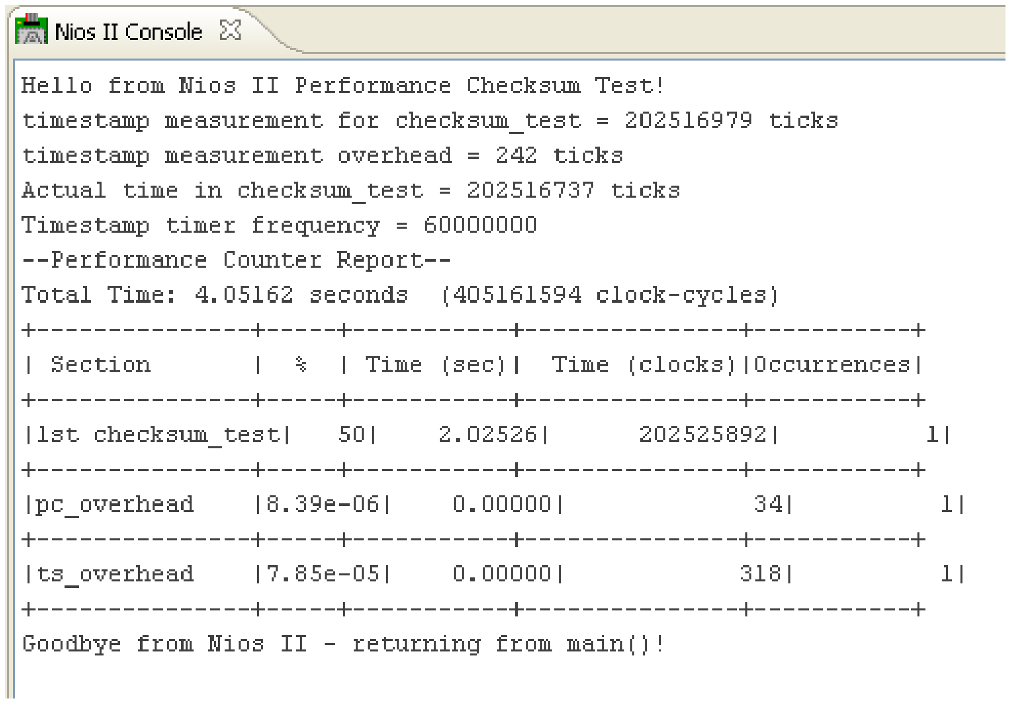
This section describes how to create and run the profiler_performance_counter software example using the Nios II SBT for Eclipse.

1. Start the Nios II SBT for Eclipse.
2. Right-click in the Project Explorer view, point to **New**, and click **Nios II Application and BSP from template**.
3. Set **SOPC Information File name** by browsing to the **standard_perf_counter** directory and selecting the .**sopcinfo** file.
4. Give your project a name, for example profiler_performance_counter.
5. Under **Templates**, select **Blank Project**.
6. Click **Finish** to create your new project.

7. Locate the <*profiler_software_examples*>**/source/SBT_flow/profiler_performance_counter** folder. Add all the files in this directory to your application project folder. For example, drag and drop the files from a file management tool to the Project Explorer view in the Nios II SBT for Eclipse.
8. Right-click your project in the Project Explorer view, point to **Nios II** and click **BSP Editor**.
9. In the Nios II BSP Editor, turn on `hal.enable_gprof` to enable the GNU profiler in your project.
10. Ensure that `hal.sys_clk_timer` is set to the **sys_clk_timer** component.
11. Set `hal.timestamp_timer` to the **high_res_timer** component.
12. Save and generate your BSP project.
13. Build your project in Nios II SBT for Eclipse.
14. To run the profiler_performance_counter software, right-click your application project, point to **Run As** and click **Nios II Hardware**.

Figure 7 illustrates the Nios II Console output after running profiler_performance_counter. As you can see, the data are very similar to the command-line example in Figure 6. Refer to “Analyzing the Performance Counter Report” for help in understanding the information in the report.

Figure 7. Performance Counter Report on Nios II Console



```
Hello from Nios II Performance Checksum Test!
timestamp measurement for checksum_test = 202516979 ticks
timestamp measurement overhead = 242 ticks
Actual time in checksum_test = 202516737 ticks
Timestamp timer frequency = 60000000
--Performance Counter Report--
Total Time: 4.05162 seconds  (405161594 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|1st checksum_test|   50|    2.02526|      202525892|          1|
+---------------+-----+-----------+---------------+-----------+
|pc_overhead    |8.39e-06|    0.00000|             34|          1|
+---------------+-----+-----------+---------------+-----------+
|ts_overhead    |7.85e-05|    0.00000|            318|          1|
+---------------+-----+-----------+---------------+-----------+
Goodbye from Nios II - returning from main()!
```

Analyzing the Performance Counter Report

The information in this section is applicable to the performance counter report regardless whether it is generated from the command line or through the Nios II SBT for Eclipse.

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

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

Conclusion

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

Troubleshooting

The following sections describe several problems that might occur, and suggest ways to deal with them.

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

`basic-block-count` information is not tracked, so switches such as `–annotated-source` do not work.

Writing to the Registers of a Nonexistent Section Counter

The performance counter report in Example 2 shows what happens when you attempt to use a nonexistent section counter of the performance counter component.

Example 2. Result of Using a Nonexistent Section Counter

```
--Performance Counter Report--
Total Time: 5.78751 seconds (289375582 clock-cycles)
+--------------------+--------+-------------+---------------+-----------+
|      Section       |   %    | Time (sec)  | Time (clocks) |Occurrences|
+--------------------+--------+-------------+---------------+-----------+
|sleep_tests         |    49.4|      2.86162|      143081026|          1|
+--------------------+--------+-------------+---------------+-----------+
|perf_begin_overhead | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|timestamp_overhead  | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|non_existent_counter|6.37e+12|368934881474.19104|        -1| 4294967295|
+--------------------+--------+-------------+---------------+-----------+
```

Suppose that a fourth section counter is specified for a performance counter component that is defined in SOPC Builder to have only three section counters (the default value).

In Example 2, the test is performed on a hardware design that does not have any other component defined with registers mapped immediately after the performance counter component's registers. Therefore, no other component is impacted. Depending on how the component register base addresses are configured in SOPC Builder for a particular hardware design, unpredictable system behavior could occur.

Output From a printf() or perf_print_formatted_output() Call Near the End of main() Might Be Prematurely Truncated

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

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

Fitting a Performance Counter in a Hardware Design That Consumes Most of an FPGA's Resources

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

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

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

If no system timer is defined for the system, the `nios2_pcsample()` function is not called, and the histogram for the **gmon.out** file is not produced. Define a system timer for your system.

Further Reading

For information about the GNU profiler, **gprof**, refer to the Nios II GNU profiler documentation, included with the GCC documentation, available at the Nios II Embedded Design Suite Support page of the Altera website.

Because Altera has rewritten the `lib-gprof` library, the information in this manual about how data is collected deviates somewhat from Altera's implementation

For information about the performance counter, refer to the *Performance Counter Core* chapter in the *Embedded Peripherals IP User Guide*. For information about the high-speed timer, refer to the *Timer Core* chapter in the *Embedded Peripherals IP User Guide*.

Referenced Documents

This application note references the following documents:

- *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*
- *Introduction to the Quartus II Software*
- *Performance Counter Core* chapter in the *Embedded Peripherals IP User Guide*
- *Timer Core* chapter in the *Embedded Peripherals IP User Guide*

Document Revision History

Table 1 shows the revision history for this application note.

Table 1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
May 2010 v2.0	This revision incorporates the following changes: ■ Added the Nios II SBT for Eclipse flow ■ Updated examples for the NEEK	Updated document, software and screen shots for the Nios II SBT for Eclipse
July 2008 v1.3	This revision incorporates the following changes: ■ Replaced references to the Nios II IDE with instructions in the Nios II software build flow. ■ General updates for the Quartus II software v8.0.	Updated document for the Quartus II software and Nios II EDS v8.0.
February 2006 v1.2	—	Updated document for the Quartus II software and Nios II EDS v5.1 SP1.
November 2005 v1.1	—	Updated document for the Quartus II software and Nios II EDS v5.1.
August 2005 v1.0	Initial release.	—

101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support
www.altera.com/support