Profiling Nios II Systems

July 2008, ver. 1.3 Application Note 391

Introduction

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 peripheral, and the performance counter peripheral. Two tutorials give detailed examples of using these tools to measure performance in the Altera® Nios II software build tools 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 peripheral and the timestamp peripheral are minimally intrusive hardware methods for measuring the performance of a Nios II system. The application note describes and compares the two peripherals. To use these methods, you add the hardware peripherals to your system, and you add source code changes to start and stop the peripherals. The hardware peripherals 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.

Tools

The tutorials assume you are familiar with the Nios II software build tools development flow for Nios II systems, including use of the Quartus® II software and SOPC Builder. The tutorials use the following tools to measure the performance of a Nios II system:

- GNU Profiler
- Performance Counter Peripheral
- High Resolution Timer

An additional tool, program counter trace collection, is available for some Nios II processors. This method 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.

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. In the Nios II software build tools flow, enable the profiler in your project by turning on the HAL settings for `enable_gprof` and `enable_exit`.

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 board support package and the application project.

Performance Counter Peripheral

A performance counter peripheral is just a block of big counters in the hardware that measure the execution time taken by the blocks of code that you choose. Each block of interest is called a code section. A performance counter peripheral can track as many as seven measured code sections. By default, the peripheral 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 blocks of interest in your program, the measured code sections. The performance counter peripheral has as many as 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.

High Resolution Timer

A high resolution timer, in contrast to a performance counter component, does not use a large number of logic elements (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 peripheral macros.

Program Counter Trace Collection

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 the *Lauterbach Trace32 Debugger and PowerTrace Hardware* section in the *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*.

Use 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 Advantage

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 Drawback

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 peripheral 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 board support package (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 board support package. 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 GNU profiler documentation at **$SOPC_KIT_NIOS2\documents\gnu-tools\binutils\gprof.html**.

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 to the `.text` section, you can compare the `.text` section 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 board support package.

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 peripherals are required. If your Nios II hardware design already includes a system timer component, you do not need to change the design.

Tutorial 1: Use the GNU Profiler to Measure Code Performance

For the first tutorial, use the example standard hardware design without modification. If your Nios development board contains another hardware design, follow the next few steps to program the standard hardware design. If the Nios development board already has the standard hardware design programmed, proceed to "Create the profiler_gnu Software Design".

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

1. Create a copy of the standard hardware design directory for your development board in a new working directory (<*project_directory*>).

   For example, the standard Verilog HDL hardware design directory for the Nios II Development Kit, Cyclone® II Edition is located at **$SOPC_KIT_NIOS2\examples\verilog\ niosII_cycloneII_2c35\standard**. To use this design, copy this directory to another named <*project_directory*>.

2. Start the Quartus II software, version 8.0 or later.

3. In the Quartus II window, on the New menu, click **Open Project**.

4. Open your new copy of the Quartus II project file for the standard Nios II hardware design for your board.

5. On the Tools menu, click **Programmer**.

6. Turn on **Program/Configure,** located on the same row as your **standard.sof** file, <*your board*>**_standard.sof**.

7. Click **Start** to download the Nios II SRAM Object File <*your board*>**_standard.sof** 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.

Create the profiler_gnu Software Design

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

1. Open a Nios II command shell.

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

2. Change to the working directory for your hardware and software projects (<*project_directory*>).

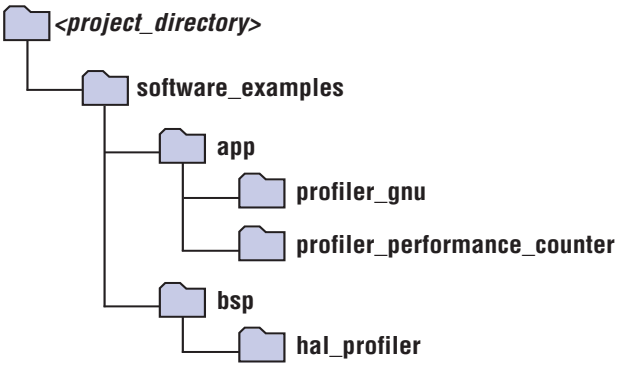
3. Ensure that the working directory <*project_directory*> and all of its subdirectories are write-enabled, by typing the following command:

   ```
   chmod -R +w . ↵
   ```

4. Download the file **profiler_software_examples.zip** to <*project_directory*>. This **.zip** file can be found on the Altera literature pages with this application note, at www.altera.com/literature/lit-nio2.jsp.

5. Unzip the file **profiler_software_examples.zip**. The directory structure shown in Figure 1 appears in <*project_directory*>.

Figure 1. Directory Structure After Unzipping Files



6. Change to the directory **software_examples/app/profiler_gnu**.

7. 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()`.

Run the profiler_gnu Software Design

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:

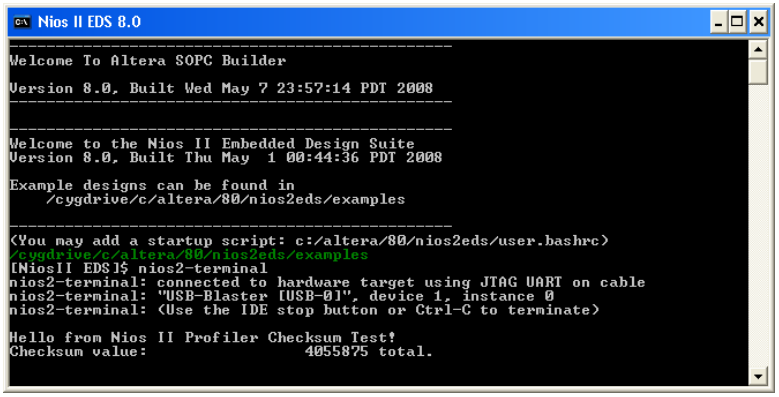
```
nios2-terminal ↵
```

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

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

The GNU profiler writes the data to the **gmon.out** file when the application calls the `exit()` function. Figure 2 shows an example of the output on the nios2-terminal window.

Figure 2. GNU Profiler Output on nios2-terminal



Create the Profiler Report Based on the profiler_gnu Design

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.

Analyze the Profiler Report

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 full GNU profiler documentation.

The profiler report excerpts shown in Example 1 were generated on a Nios development board, Cyclone II Edition, containing a Cyclone II 2C35 device with a Nios II version 8.0 standard hardware design running at 85 MHz.

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
94.01   9.53    9.53          1      9.53     9.58  checksum_test_routine
 3.85   9.92     0.39         9      0.04     0.04      alt_busy_sleep
 1.28  10.05     0.13                                  alt_dcache_flush
... (deleted portion) ...


Call graph (explanation follows)

granularity: each sample hit covers 32 byte(s) for 0.10% of 10.14 seconds

index % time    self  children    called     name
                9.53    0.05       1/1        main [3]
[4]     94.5    9.53    0.05       1          checksum_test_routine [4]
                0.00    0.00     300/300      alt_dcache_flush_all [25]
... (deleted portion) ...


                0.00    0.00    1024/1024     alt_irq_handler [23]
[22]     0.0    0.00    0.00    1024          alt_avalon_timer_sc_irq [22]
```

In Example 1, the call graph report shows that the `checksum_test_routine()` function call (index [4]) consumed 94.5% of the processing time during execution of the profiler_project design.

The granularity statement in the call graph report states that the report covers 10.14 seconds, or 10,140 milliseconds. Our Nios II system has a 10 millisecond timer, so the timer interrupt handler is called once at the beginning before a full clock period has elapsed and once every 10 milliseconds thereafter. A precise report would show, therefore, that the timer interrupt handler is called 1014 times. Index [22] shows that `alt_avalon_timer_sc_irq` is called 1024 times, which is within the precision range of this measurement method.

Use 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 1S40 device, a single

performance counter peripheral with three section counters defined within a modified standard hardware design consumes 670 logic cells (LCs), and 420 LC registers. In the same design, a single performance counter defined with seven section counters consumes 1,345 logic cells and 808 LC registers.

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 $\mathbf{f_{MAX}}$.

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 multi-threading context switching, although it is much more pronounced in a multi-threaded system. Many threads or interrupt service routines may 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 block you are attempting to measure.

To avoid context switch impacts, most multi-threaded 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 peripheral macros to record the beginning and ending times of each measured section.

PERF_BEGIN and PERF_END are single writes to the performance counter peripheral. 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 "Performance Counter Peripheral" on page 2. 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 Volume 5: *Embedded Peripherals* of the *Quartus II Handbook*.

Hardware Considerations

Performance counters and timestamp interval timers are SOPC Builder peripherals. 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 2: Use Performance Counters and Timers to Measure Code Performance

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 create the standard_perf_counter design, by modifying the standard example to change the frequency of the interval timer and to add the performance counter.

Alternatively, you can implement the software part of this tutorial without creating the standard_perf_counter Nios II hardware design, by using the full-featured hardware example design provided with your Nios II installation. For information about how to use the full-featured design in this tutorial, refer to "Appendix A: Full-Featured Reference Design" on page 21.

Create the standard_perf_counter Hardware Design

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

1. Create a copy of the standard hardware design for your Nios development board.

   For example, the standard Verilog HDL example design for the Nios II Development Kit, Cyclone II Edition is located in the directory **$SOPC_KIT_NIOS2\examples\verilog\niosII_cycloneII_2c35\standard**. To use this design, copy this directory to another named **standard_perf_counter**.

1. Start the Quartus II software, version 8.0 or later.

2. In the Quartus II window, on the New menu, click **Open Project**.

3. Open the Quartus II project file for the hardware design you just copied. For example, the standard project file for the Cyclone II development board is the file **NiosII_cycloneII_2c35_standard.qpf**, located in your new directory **standard_perf_counter**.

4. On the Tools menu, click **SOPC Builder**.

   The SOPC Builder window appears.

5. In the **System Contents** tab, under **Peripherals**, expand the **Debug and Performance** category.

6. Click **Performance Counter Unit**.

7. Click **Add**.

8. Leave the default value of **Number of simultaneously-measured sections** at 3.

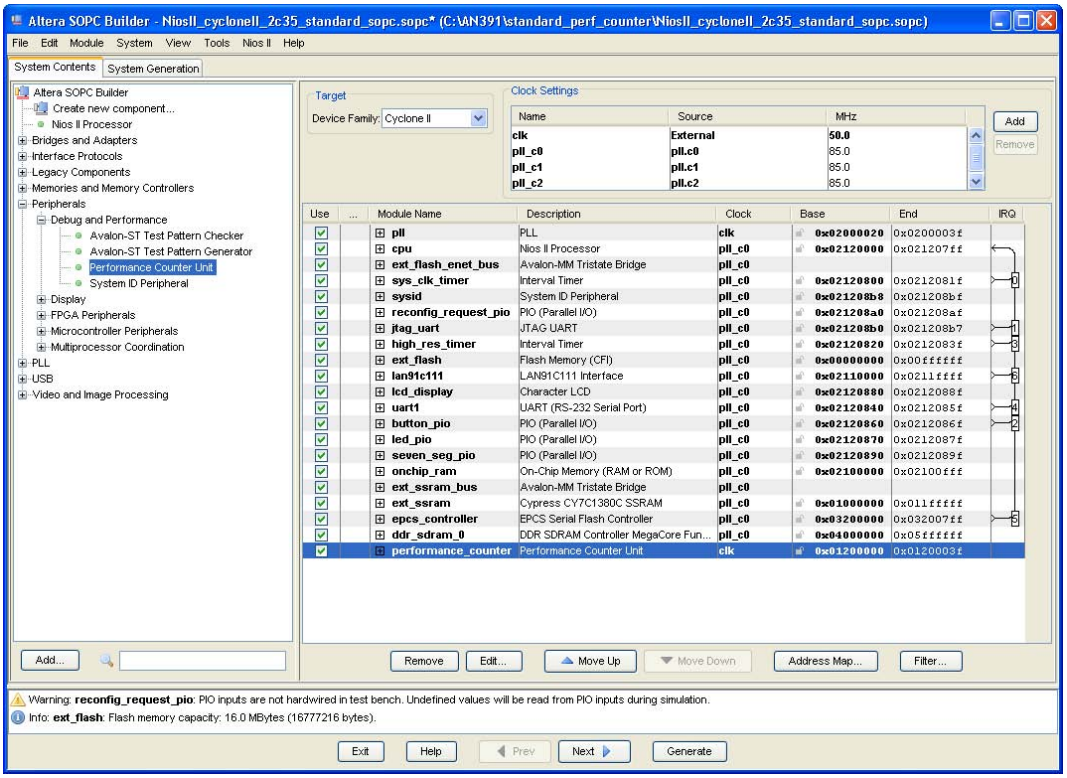
9. Click **Finish**.

   A performance counter module is added to the hardware design.

10. Under the list of Module Names that make up the hardware design, select the interval timer named **high_res_timer**.
11. Right-click **high_res_timer** and click **Edit**.
12. Under **Timeout Period**, set the **Initial Period** value number to `1` and the units to **usec** (microseconds).
13. Click **Finish**.

    Figure 3 shows the SOPC Builder system.

Figure 3. SOPC Builder Window



14. Click the **System Generation** tab.
15. Click **Generate**. The generation phase takes a few minutes.

16. The final message should state "SUCCESS: SYSTEM GENERATION COMPLETED". When system generation is complete, click **Exit**. The hardware design is now ready to be compiled by the Quartus II software.

17. In the Processing menu, click **Start Compilation**.

18. When you see the message "Full Compilation was successful", click **OK**. This step generates the file <*your_board*>**_standard.sof**.

Program the standard_perf_counter Hardware Design to an FPGA

Now you can program your new hardware design in the FPGA, by performing the following steps:

1. On the Tools menu, click **Programmer**.

2. Turn on **Program/Configure,** located on the same row as your **standard.sof** file, <*your board*>**_standard.sof**.

3. Click **Start** to download <*your board*>**_standard.sof** 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.

Create the profiler_performance_counter Software Design

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.

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

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. Download the file **profiler_software_examples.zip** to <*project_directory*>. This **.zip** file can be found on the Altera literature pages with this application note, at www.altera.com/literature/lit-nio2.jsp.

5. Unzip the file **profiler_software_examples.zip**. The directory structure shown in Figure 1 on page 9 appears in <*project_directory*>.

6. Change to the directory **software_examples/app/profiler_performance_counter**.

7. 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 <*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()`.

8. Edit the **parameter_definition.tcl** file by adding the following two lines before the two lines described in Step 7:

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

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

Run the profiler_performance_counter Software Design

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:

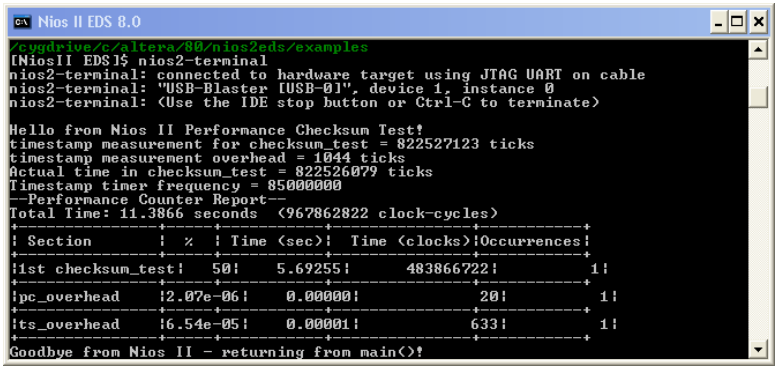
   ```
   nios2-terminal ↵
   ```

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 the nios2-terminal, by typing the following command:

```
nios2-download -g *.elf ↵
```

Figure 4 shows an example of the output that appears in the nios2-terminal window. Your output may vary.

Figure 4. Performance Counter Report on nios2-terminal



`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` will not work.

Writing to the Registers of a Non-Existent Section Counter

The performance counter report in Example 2 shows what happens when you attempt to use a non-existent section counter of the performance counter peripheral.

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

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

Example 2. Result of Using a Non-Existent 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|
+--------------------+--------+-------------+---------------+-----------+
```

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

This 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

The system could be measured in a larger FPGA for development 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 documentation located at **$SOPC_KIT_NIOS2\documents\gnu-tool\gprof.html**. 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 Volume 5: *Embedded Peripherals* of the *Quartus II Handbook*.

For information about the high-speed timer, refer to the *Timer Core* chapter in Volume 5: *Embedded Peripherals* of the *Quartus II Handbook*.

Appendix A: Full-Featured Reference Design

This section describes the steps to run the profiler_performance_counter software project on the full_featured Nios II hardware design.

To open the project file, perform the following steps:

1. Create a copy of the full-featured hardware design for your Nios development board.

For example, the full-featured Verilog HDL example design for the Nios II Development Kit, Cyclone II Edition is located in the directory **$SOPC_KIT_NIOS2\examples\verilog\niosII_cycloneII_2c35\full_featured**. To use this design, copy this directory to another named **<***project_directory***>**.

2. Start the Quartus II software, version 8.0 or later.

3. In the Quartus II window, on the New menu, click **Open Project**.

4. Open the Quartus II project file for the full-featured Nios II hardware design project for your board. For example, the full-featured Verilog HDL project file for the Cyclone II development board is the file **NiosII_cycloneII_2c35_full_featured.qpf**, located in the directory **$SOPC_KIT_NIOS2\examples\verilog\niosII_cycloneII_2c35\full_featured**.

Program the Full-Featured Hardware Design to an FPGA

Now you can program the full-featured hardware design in the FPGA, by performing the following steps:

1. On the Tools menu, click **Programmer**.

2. Turn on **Program/Configure**, located on the same row as your **full_featured.sof** file, <*your board*>**_full_featured.sof**.

3. Click **Start** to download <*your board*>**_full_featured.sof** to the FPGA.

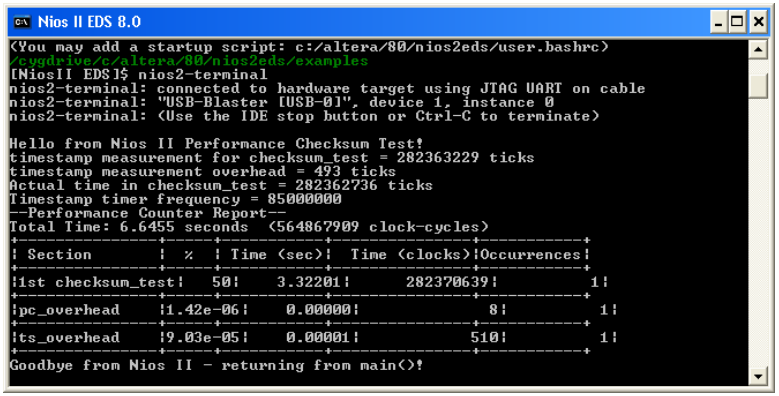
If the **Start** button is greyed out, or the USB-Blaster cable is not listed in the **Hardware Setup...** field, refer to the *Introduction to the Quartus II Software* manual for more details on the Programmer tool.

Create and Run the profiler_performance_counter Software Design

Create and run a software project to test the full-featured hardware design by performing the Tutorial 2 steps in "Create the profiler_performance_counter Software Design" on page 17 and in "Run the profiler_performance_counter Software Design" on page 18.

The output shown in Figure 5 appears in the nios2-terminal window.

Figure 5. Full-Featured Example Performance Output



`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.

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 Volume 5: *Embedded Peripherals* of the *Quartus II Handbook*
- *Timer Core* chapter in Volume 5: *Embedded Peripherals* of the *Quartus II Handbook*

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
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/