29. Performance Counter Core

QII55001-9.0.0

Core Overview

The performance counter core with Avalon® interface enables relatively unobtrusive, real-time profiling of software programs. With the performance counter, you can accurately measure execution time taken by multiple sections of code. You need only add a single instruction at the beginning and end of each section to be measured.

The main benefit of using the performance counter core is the accuracy of the profiling results. Alternatives include the following approaches:

- GNU profiler, `gprof`—`gprof` provides broad low-precision timing information about the entire software system. It uses a substantial amount of RAM, and degrades the real-time performance. For many embedded applications, `gprof` distorts real-time behavior too much to be useful.
- Interval timer peripheral—The interval timer is less intrusive than `gprof`. It can provide good results for narrowly targeted sections of code.

The performance counter core is unobtrusive, requiring only a single instruction to start and stop profiling, and no RAM. It is appropriate for high-precision measurements of narrowly targeted sections of code.

For further discussion of all three profiling methods, refer to *AN 391: Profiling Nios II Systems*.

The performance counter core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. The core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera® device drivers enable the Nios II processor to use the performance counters.

This chapter contains the following sections:

Functional Description

The performance counter core is a set of counters which track clock cycles, timing multiple sections of your software. You can start and stop these counters in your software, individually or as a group. You can read cycle counts from hardware registers.

The core contains two counters for every section:

- Time: A 64-bit clock cycle counter.
- Events: A 32-bit event counter.

Section Counters

Each 64-bit time counter records the aggregate number of clock cycles spent in a section of code. The 32-bit event counter records the number of times the section executes.

The performance counter core can have up to seven section counters.

Global Counter

The global counter controls all section counters. The section counters are enabled only when the global counter is running.

The 64-bit global clock cycle counter tracks the aggregate time for which the counters were enabled. The 32-bit global event counter tracks the number of global events, that is, the number of times the performance counter core has been enabled.

Register Map

The performance counter core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to memory-mapped registers. Reading from the registers retrieves the current times and event counts. Writing to the registers starts, stops and resets the counters. Table 29–1 shows the registers in detail.

Table 29–1. Performance Counter Core Register Map (Part 1 of 2)

Offset	Register Name	Bit Description		
		Read	Write	
		31 ... 0	31 ... 1	0
0	T[0]$_{lo}$	global clock cycle counter [31: 0]	*(1)*	0 = STOP 1 = RESET
1	T[0]$_{hi}$	global clock cycle counter [63:32]	*(1)*	0 = START
2	Ev[0]	global event counter	*(1)*	*(1)*
3	—	*(1)*	*(1)*	*(1)*
4	T[1]$_{lo}$	section 1 clock cycle counter [31:0]	*(1)*	0 = STOP
5	T[1]$_{hi}$	section 1 clock cycle counter [63:32]	*(1)*	0 = START
6	Ev[1]	section 1 event counter	*(1)*	*(1)*
7	—	*(1)*	*(1)*	*(1)*
8	T[2]$_{lo}$	section 2 clock cycle counter [31:0]	*(1)*	0 = STOP

Table 29–1. Performance Counter Core Register Map (Part 2 of 2)

Offset	Register Name	Bit Description		
		Read	Write	
		31 ... 0	31 ... 1	0
9	$T[2]_{hi}$	section 2 clock cycle counter [63:32]	*(1)*	0 = START
10	Ev[2]	section 2 event counter	*(1)*	*(1)*
11	—	*(1)*	*(1)*	*(1)*
. . .	. . .	. . .	. . .	. . .
4*n* + 0	$T[n]_{lo}$	section *n* clock cycle counter [31:0]	*(1)*	0 = STOP
4*n* + 1	$T[n]_{hi}$	section *n* clock cycle counter [63:32]	*(1)*	0 = START
4*n* + 2	Ev[n]	section *n* event counter	*(1)*	*(1)*
4*n* + 3	—	*(1)*	*(1)*	*(1)*

Note to Table 29–1:

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

System Reset Considerations

After system reset, the performance counter core is stopped and disabled, and all counters contain zero.

Device and Tools Support

The performance counter core supports all Altera device families supported by SOPC Builder, and provides device drivers for the Nios II hardware abstraction layer (HAL) system library.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the performance counter core in SOPC Builder to specify the core's hardware features.

Define Counters

Choose the number of section counters you want to generate by selecting from the **Number of simultaneously-measured sections** list. The performance counter core may have up to seven sections. If you require more that seven sections, you can instantiate multiple performance counter cores.

Multiple Clock Domain Considerations

If your SOPC Builder system uses multiple clocks, place the performance counter core in the same clock domain as the CPU. Otherwise, it is not possible to convert cycle counts to seconds correctly.

Hardware Simulation Considerations

You can use this core in simulation with no special considerations.

Software Programming Model

The following sections describe the software programming model for the performance counter core.

Software Files

Altera provides the following software files for Nios II systems. These files define the low-level access to the hardware and provide control and reporting functions. Do not modify these files.

- **altera_avalon_performance_counter.h, altera_avalon_performance_counter.c**—The header and source code for the functions and macros needed to control the performance counter core and retrieve raw results.
- **perf_print_formatted_report.c**—The source code for simple profile reporting.

Using the Performance Counter

In a Nios II system, you can control the performance counter core with a set of highly efficient C macros, and extract the results with C functions.

API Summary

The Nios II application program interface (API) for the performance counter core consists of functions, macros and constants.

Functions and macros

Table 29–2 lists macros and functions for accessing the performance counter hardware structure.

Table 29–2. Performance Counter Macros and Functions

Name	Summary
`PERF_RESET()`	Stops and disables all counters, resetting them to 0.
`PERF_START_MEASURING()`	Starts the global counter and enables section counters.
`PERF_STOP_MEASURING()`	Stops the global counter and disables section counters.
`PERF_BEGIN()`	Starts timing a code section.
`PERF_END()`	Stops timing a code section.
`perf_print_formatted_report()`	Sends a formatted summary of the profiling results to `stdout`.
`perf_get_total_time()`	Returns the aggregate global profiling time in clock cycles.
`perf_get_section_time()`	Returns the aggregate time for one section in clock cycles.
`perf_get_num_starts()`	Returns the number of counter events.
`alt_get_cpu_freq()`	Returns the CPU frequency in Hz.

For a complete description of each macro and function, see "Performance Counter API" on page 29–6.

Hardware Constants

You can get the performance counter hardware parameters from constants defined in **system.h**. The constant names are based on the performance counter instance name, specified on the **System Contents** tab in SOPC Builder. Table 29–3 lists the hardware constants.

Table 29–3. Performance Counter Constants

Name *(1)*	Meaning
PERFORMANCE_COUNTER_BASE	Base address of core
PERFORMANCE_COUNTER_SPAN	Number of hardware registers
PERFORMANCE_COUNTER_HOW_MANY_SECTIONS	Number of section counters

Note to Table 29–3:

(1) Example based on instance name `performance_counter`.

Startup

Before using the performance counter core, invoke `PERF_RESET` to stop, disable and zero all counters.

Global Counter Usage

Use the global counter to enable and disable the entire performance counter core. For example, you might choose to leave profiling disabled until your software has completed its initialization.

Section Counter Usage

To measure a section in your code, surround it with the macros `PERF_BEGIN()` and `PERF_END()`. These macros consist of a single write to the performance counter core.

You can simultaneously measure as many code sections as you like, up to the number specified in SOPC Builder. See "Define Counters" on page 29–3 for details. You can start and stop counters individually, or as a group.

Typically, you assign one counter to each section of code you intend to profile. However, in some situations you may wish to group several sections of code in a single section counter. As an example, to measure general interrupt overhead, you can measure all interrupt service routines (ISRs) with one counter.

To avoid confusion, assign a mnemonic symbol for each section number.

Viewing Counter Values

Library routines allow you to retrieve and analyze the results. Use `perf_print_formatted_report()` to list the results to `stdout`, as shown in Example 29–1.

Example 29–1.

```
perf_print_formatted_report(
     (void *)PERFORMANCE_COUNTER_BASE, // Peripheral's HW base address
     alt_get_cpu_freq(),               // defined in "system.h"
     3,                                // How many sections to print
     "1st checksum_test",              // Display-names of sections
     "pc_overhead",
     "ts_overhead");
```

Example 29–2 creates a table similar to this result.

Example 29–2.

```
--Performance Counter Report--
Total Time: 2.07711 seconds (103855534 clock-cycles)
+-----------------+--------+-----------+---------------+-----------+
| Section         |    %   | Time (sec)| Time (clocks) |Occurrences|
+-----------------+--------+-----------+---------------+-----------+
|1st checksum_test|     50 |   1.03800 |      51899750 |         1 |
+-----------------+--------+-----------+---------------+-----------+
| pc_overhead     |1.73e-05|   0.00000 |            18 |         1 |
+-----------------+--------+-----------+---------------+-----------+
| ts_overhead     |4.24e-05|   0.00000 |            44 |         1 |
+-----------------+--------+-----------+---------------+-----------+
```

For full documentation of `perf_print_formatted_report()`, see "Performance Counter API" on page 29–6.

Interrupt Behavior

The performance counter core does not generate interrupts.

You can start and stop performance counters, and read raw performance results, in an interrupt service routine (ISR). Do not call the `perf_print_formatted_report()` function from an ISR.

If an interrupt occurs during the measurement of a section of code, the time taken by the CPU to process the interrupt and return to the section is added to the measurement time. The same applies to context switches in a multithreaded environment. Your software must take appropriate measures to avoid or handle these situations.

Performance Counter API

This section describes the application programming interface (API) for the performance counter core.

For Nios II processor users, Altera provides routines to access the performance counter core hardware. These functions are specific to the performance counter core and directly manipulate low level hardware. The performance counter core cannot be accessed via the HAL API or the ANSI C standard library.

PERF_RESET()

Prototype:	`PERF_RESET(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	p—performance counter core base address.
Returns:	—
Description:	Macro `PERF_RESET()` stops and disables all counters, resetting them to 0.

PERF_START_MEASURING()

Prototype:	`PERF_START_MEASURING(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	p—performance counter core base address.
Returns:	—
Description:	Macro `PERF_START_MEASURING()` starts the global counter, enabling the performance counter core. The behavior of individual section counters is controlled by `PERF_BEGIN()` and `PERF_END()`. `PERF_START_MEASURING()` defines the start of a global event, and increments the global event counter. This macro is a single write to the performance counter core.

PERF_STOP_MEASURING()

Prototype:	`PERF_STOP_MEASURING(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	p—performance counter core base address.
Returns:	—
Description:	Macro `PERF_STOP_MEASURING()` stops the global counter, disabling the performance counter core. This macro is a single write to the performance counter core.

PERF_BEGIN()

Prototype:	`PERF_BEGIN(p,n)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	p—performance counter core base address. n—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.

Returns: —

Description: Macro `PERF_BEGIN()` starts the timer for a code section, defining the beginning of a section event, and incrementing the section event counter. If you subsequently use `PERF_STOP_MEASURING()` and `PERF_START_MEASURING()` to disable and re-enable the core, the section counter will resume. This macro is a single write to the performance counter core.

PERF_END()

Prototype: `PERF_END(p,n)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: p—performance counter core base address.

n—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.

Returns: —

Description: Macro `PERF_END()` stops timing a code section. The section counter does not run, regardless whether the core is enabled or not. This macro is a single write to the performance counter core.

perf_print_formatted_report()

Prototype:
```
int perf_print_formatted_report (
    void* perf_base,
    alt_u32 clock_freq_hertz,
    int num_sections,
    char* section_name_1, ...
    char* section_name_n)
```

Thread-safe: No.

Available from ISR: No.

Include: **<altera_avalon_performance_counter.h>**

Parameters:

`perf_base`—Performance counter core base address.

`clock_freq_hertz`—Clock frequency.

`num_sections`—The number of section counters to display. This must not exceed *<instance_name>*_HOW_MANY_SECTIONS.

`section_name_1 ... section_name_n`—The section names to display. The number of section names varies depending on the number of sections to display.

Returns: 0

Description: Function `perf_print_formatted_report()` reads the profiling results from the performance counter core, and prints a formatted summary table.

This function disables all counters. However, for predictable results in a multi-threaded or interrupt environment, invoke `PERF_STOP_MEASURING()` when you reach the end of the code to be measured, rather than relying on `perf_print_formatted_report()`.

This function requires the C standard library. Do not use the small C library with this function.

perf_get_total_time()

Prototype: `alt_u64 perf_get_total_time(void* hw_base_address)`

Thread-safe: No.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—base address of performance counter core.

Returns: Aggregate global time in clock cycles.

Description: Function `perf_get_total_time()` reads the raw global time. This is the aggregate time, in clock cycles, that the performance counter core has been enabled. This function has the side effect of stopping the counters.

perf_get_section_time()

Prototype: `alt_u64 perf_get_section_time (void* hw_base_address, int which_section)`

Thread-safe: No.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—performance counter core base address.

`which_section`—counter section number.

Returns: Aggregate section time in clock cycles.

Description: Function `perf_get_section_time()` reads the raw time for a given section. This is the time, in clock cycles, that the section has been running. This function has the side effect of stopping the counters.

perf_get_num_starts()

Prototype: `alt_u32 perf_get_num_starts (void* hw_base_address, int which_section)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—performance counter core base address.

`which_section`—counter section number.

Returns: Number of counter events.

Description: Function `perf_get_num_starts()` retrieves the number of counter events (or times a counter has been started). If `which_section` = 0, it retrieves the number of global events (times the performance counter core has been enabled). This function does not stop the counters.

alt_get_cpu_freq()

Prototype: `alt_u32 alt_get_cpu_freq()`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters:

Returns: CPU frequency in Hz.

Description: Function `alt_get_cpu_freq()` returns the CPU frequency in Hz.

Referenced Documents

This chapter references the application note, *AN 391: Profiling Nios II Systems*.

Document Revision History

Table 29–4 shows the revision history for this chapter.

Table 29–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Updated the parameter description of the function `perf_print_formatted_report()`.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.