AN595: Vectored Interrupt Controller Usage and Applications

© November 2009 AN-595-1.0

The ability to process interrupt events quickly and to handle large numbers of interrupts can be critical to many embedded systems. The Vectored Interrupt Controller (VIC) is designed to address these requirements. The VIC can provide interrupt performance four to five times better than the Nios® II processor's default internal interrupt controller (IIC). The VIC also allows expansion to a virtually unlimited number of interrupts, through daisy chaining.

This document explains how to use the VIC in your hardware design, from both a hardware perspective and a software perspective. This document includes the following sections:

Prerequisites

A complete understanding of this document requires that you be familiar with the following topics:

- Creating systems in SOPC Builder
- The Nios II processor's external interrupt controller (EIC) interface
- The VIC core
- Developing software with the Nios II Embedded Design Suite (EDS)
- The Altera® HAL interrupt application programming interfaces (APIs)
- Creating and building software projects with the Nios II Software Build Tools

For information about SOPC Builder, refer to *Volume 4: SOPC Builder* in the *Quartus II Handbook*. For information about the Nios II processor's EIC interface, refer to the *Processor Architecture* and *Programming Model* chapters of the *Nios II Processor Reference Handbook*. For information about the VIC core, refer to the *Vectored Interrupt Controller Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*. For information about the Nios II EDS, including the interrupt APIs and the Software Build Tools, refer to the *Nios II Software Developer's Handbook*.

Overview of VIC Hardware

This section describes the hardware components required in a Nios II system using the VIC.

External Interrupt Controller Interface

The VIC is Altera's implementation of an EIC. An EIC is an interrupt controller implemented as a component separate from the Nios II processor core. An EIC provides a higher-performance alternative to the Nios II processor's IIC.

The EIC interface is a feature that you can add to your Nios II processor in SOPC Builder. The EIC interface includes an Avalon Streaming (Avalon-ST) sink, allowing an EIC to communicate interrupt information to the processor. The EIC interface's Avalon-ST sink also allows the connected EIC to relay interrupt information from daisy-chained EICs.

Although this application note discusses only the VIC, the EIC interface on the Nios II processor is designed to support custom EICs, as well.

For general information about the EIC interface, refer to the *Processor Architecture* and *Programming Model* chapters of the *Nios II Processor Reference Handbook.*

Shadow Register Sets

You can add shadow register sets to the Nios II processor. Used in conjunction with the VIC, shadow register sets eliminate much of the time normally spent switching between exception context and application context. HAL interrupt support for EICs requires at least one shadow register set.

Vectored Interrupt Controller

The VIC offers high-performance, low-latency interrupt handling. The VIC prioritizes interrupts in hardware and outputs information about the highest-priority pending interrupt.

The VIC works with the Nios II processor's EIC interface. The VIC is designed for hardware compatibility with any EIC in a daisy chain configuration. However, the Nios II Hardware Abstraction Layer (HAL) requires that all EICs in a daisy chain be of the same class, so that they are all supported by the same driver.

Reasons to Use the VIC

You might want to use the VIC in your hardware design for one or more of the following reasons:

- You need to reduce average response time to one or more interrupts.
- You have hard real-time requirements for interrupt performance.
- You require nonmaskable interrupts.
- You need to handle more than 32 interrupts (the maximum supported by the IIC).

Implementing the VIC in SOPC Builder

This section describes how to incorporate one or more VICs in your SOPC Builder system, and how to support the VIC in software.

Adding VIC Hardware

When you add a VIC to your SOPC Builder system, you must perform the following high-level tasks:

1. Add the EIC interface to your Nios II processor core
2. Optionally add shadow register sets to your Nios II processor core (required if you intend to use HAL interrupt support)
3. Add and parameterize one or more VIC components
4. Connect interrupt sources to the VIC component(s)

Adding the EIC Interface Shadow Register Sets

This section describes how to add the EIC interface and shadow register sets to a Nios II processor core in SOPC Builder, through the MegaWizard™ interface.

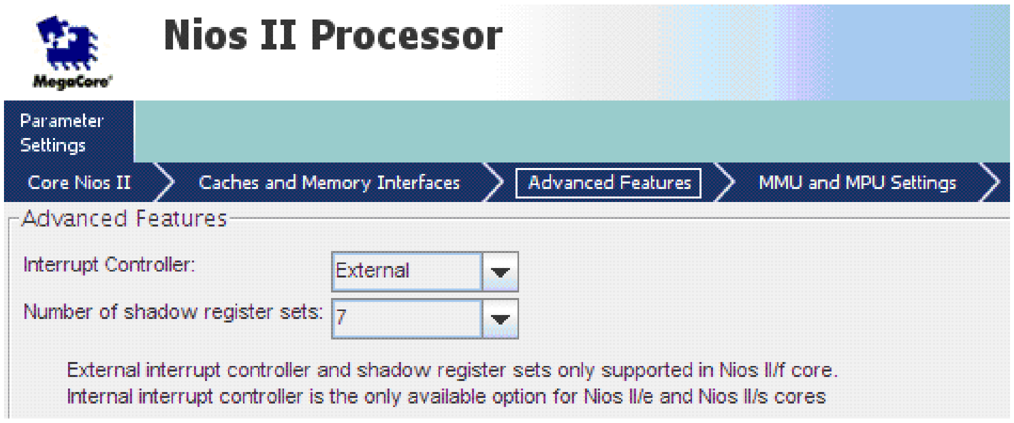
1. In SOPC Builder, double-click the Nios II processor to open the MegaWizard interface.
2. Enable the EIC interface on the Nios II processor by selecting it in the **Interrupt Controller** list in the **Advanced Features** tab, as shown in Figure 1.

   There are two options for **Interrupt Controller**: **Internal** and **External**. If you select **Internal**, the processor is implemented with the internal interrupt controller. Select **External** to implement the processor with an EIC interface.

   When you implement the EIC interface, you must connect an EIC, such as the VIC. Failure to connect an EIC results in an SOPC Builder error.

3. Select the desired number of shadow register sets. In the **Number of shadow register sets** list, select the number of register sets that matches your system performance goals.
4. Click **Finish** to exit from the Nios II MegaWizard interface. Notice that the processor shows an unconnected `interrupt_controller_in` Avalon-ST sink, as shown in Figure 2.

Figure 1. Configuring the Interrupt Controller and Shadow Register Sets

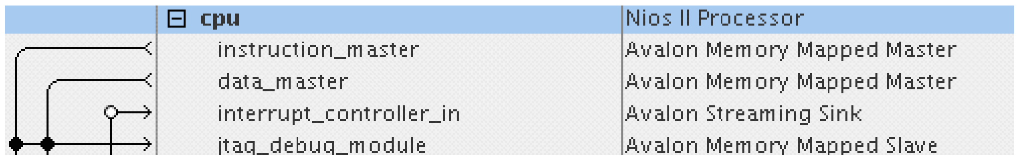


Interrupt Controller: External

Number of shadow register sets: 7

External interrupt controller and shadow register sets only supported in Nios II/f core.
Internal interrupt controller is the only available option for Nios II/e and Nios II/s cores

Figure 2. Nios II Processor with EIC Interface



cpu		Nios II Processor
	instruction_master	Avalon Memory Mapped Master
	data_master	Avalon Memory Mapped Master
	interrupt_controller_in	Avalon Streaming Sink
	jtag_debug_module	Avalon Memory Mapped Slave

Shadow register sets reduce the context switching overhead associated with saving and restoring registers, which can otherwise be significant. If possible, add one shadow register set for each interrupt that requires high performance.

VIC Instantiation, Parameterization, and Connection

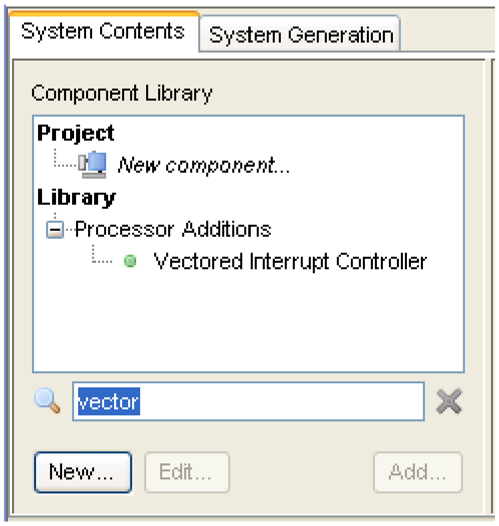
After you add the EIC interface and shadow register set(s) to the Nios II processor, you instantiate and parameterize the VIC in your SOPC Builder system.

Instantiation

To instantiate a VIC in your SOPC Builder system, execute the following steps:

1. Browse to the **Component Library** column in the **System Contents** tab of SOPC Builder.
2. Type `vector` in the search box below this column. The MegaWizard interface hides all components except the VIC, as shown in Figure 3.
3. Double click the Vectored Interrupt Controller component to add this component to your SOPC Builder System.

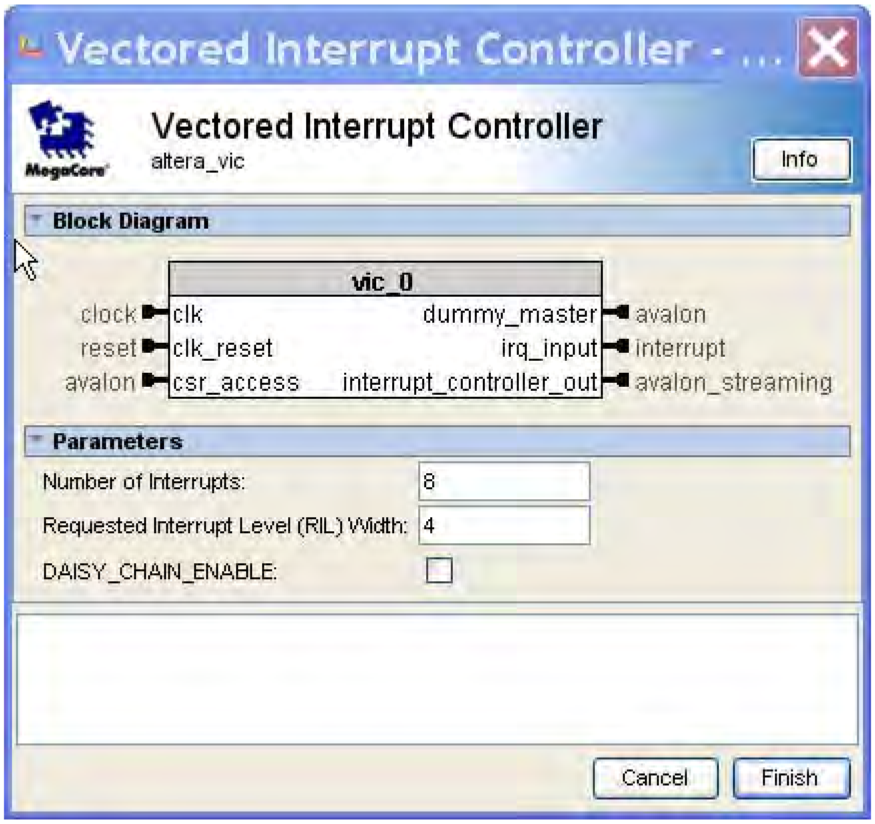
Figure 3. Vectored Interrupt Controller Component



Parameterization

When you add the VIC to your system, the **Vectored Interrupt Controller** MegaWizard interface appears as shown in Figure 4.

Figure 4. Vectored Interrupt Controller Parameterization



Number of Interrupts: 8

Requested Interrupt Level (RIL) Width: 4

DAISY_CHAIN_ENABLE: ☐

The VIC MegaWizard interface allows you to specify the following options:

- **Number of Interrupts**—The number of interrupts your VIC must support.
- **Requested Interrupt Level (RIL) Width**—The number of bits allocated to represent the interrupt level for each interrupt.

  For a full description of the RIL, including the rules for default RIL assignment, refer to the *Vectored Interrupt Controller Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

- **DAISY_CHAIN_ENABLE**—Allows the VIC to daisy chain to another EIC. Turn on this option if you want to support multiple VICs in your system.

  Study the VIC Daisy-Chain example that accompanies this document for a usage example.

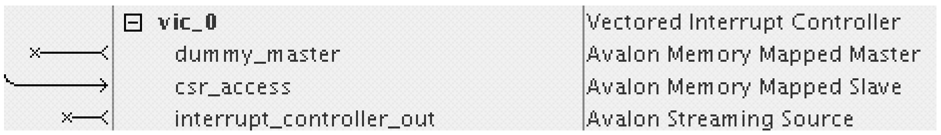
When you have finished parameterizing the VIC, click **Finish** to instantiate the component in your SOPC Builder system.

VIC Connections

When you have added the VIC to your system, it appears in SOPC Builder as shown in Figure 5.

If you have enabled daisy chaining, SOPC Builder adds an Avalon-ST sink, called `interrupt_controller_in`, to the VIC.

Figure 5. VIC Interfaces



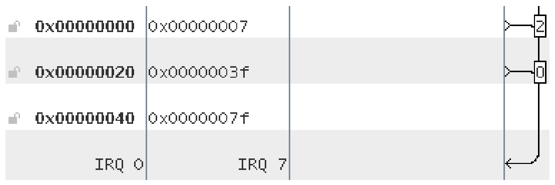
vic_0
dummy_master
csr_access
interrupt_controller_out
Vectored Interrupt Controller
Avalon Memory Mapped Master
Avalon Memory Mapped Slave
Avalon Streaming Source

After adding a VIC to the SOPC Builder system, you must parameterize the VIC and the EIC interface at the system level. Immediately after you add the VIC, several error messages appear. Resolve these error messages by executing the following actions in any order:

- Connect the VIC's `interrupt_controller_out` Avalon-ST source to the `interrupt_controller_in` Avalon-ST sink on either the Nios II processor or the next VIC in a daisy-chained configuration.
- Connect the VIC's `dummy_master` Avalon Memory-Mapped (Avalon-MM) port to the `csr_access` Avalon-MM slave port.
- Assign an interrupt number for each interrupt-based component in the system, as shown in Figure 6. This step connects each component to an interrupt port on the VIC.

  If your system contains more than one EIC connected to a single processor, you must ensure that each component is connected to an interrupt port on only one EIC.

Figure 6. Assigning Interrupt Numbers



When you use the HAL VIC driver, the driver makes a default assignment from register sets to interrupts. The default assignment makes some assumptions about interrupt priorities, based on how devices are connected to the VIC.

For details of the default assignment, including default RIL and RRL settings, refer to "Altera HAL Software Programming Model" in the *Vectored Interrupt Controller Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

To make effective use of the VIC interrupt setting defaults, assign your highest priority interrupts to low interrupt port numbers on the VIC closest to the processor.

Software

If you write an interrupt handler for a system based on the VIC component, you must use the HAL enhanced interrupt API to register the handler and control its runtime environment. The enhanced interrupt API provides a number of functions for use with EICs, including the VIC. This section describes a subset of the functions in the enhanced interrupt API.

For information about the enhanced interrupt API, refer to "Interrupt Service Routines" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

In particular, this section shows how to code a driver so that it supports both the enhanced API and the legacy API. This must include testing for the presence of the enhanced API, and conditionally calling the appropriate function.

alt_ic_isr_register() versus alt_irq_register()

The enhanced API function `alt_ic_isr_register()` is very similar to the legacy function `alt_irq_register()`, with a few important differences. The differences between these two functions are best understood by examining the code in Example 1. This example registers a timer interrupt in either the legacy API or the enhanced API, whichever is implemented in the board support package (BSP). Example 1 is taken directly from the example code accompanying this document.

Example 1. Registering an ISR with Both APIs

```
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
void timer_interrupt_latency_init (void* base, alt_u32 irq_controller_id, alt_u32 irq)
{
 /* Register the interrupt */
 alt_ic_isr_register(irq_controller_id, irq, timer_interrupt_latency_irq, base, NULL);
 /* Start timer */
 IOWR_ALTERA_AVALON_TIMER_CONTROL(base, ALTERA_AVALON_TIMER_CONTROL_ITO_MSK
 | ALTERA_AVALON_TIMER_CONTROL_START_MSK);
}
#else
void timer_interrupt_latency_init (void* base, alt_u32 irq)
{
 /* Register the interrupt */
 alt_irq_register(irq, base, timer_interrupt_latency_irq);
 /* Start timer */
 IOWR_ALTERA_AVALON_TIMER_CONTROL(base, ALTERA_AVALON_TIMER_CONTROL_ITO_MSK
 | ALTERA_AVALON_TIMER_CONTROL_START_MSK);
}
#endif
```

The first line of Example 1 detects whether the BSP implements the enhanced interrupt API. If the enhanced API is implemented, the `timer_interrupt_latency_init()` function calls the enhanced function. If not, `timer_interrupt_latency_init()` reverts to the legacy interrupt API function.

For an explanation of how the Nios II Software Build Tools select which API to implement in a BSP, refer to "Interrupt Service Routines" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Example 2 shows the function prototype for `alt_ic_isr_register()`, which registers an ISR in the enhanced API. The interrupt controller identifier (for argument `ic_id`) and the interrupt port number (for argument `irq`) are defined in **system.h**.

Example 2. Enhanced Function alt_ic_isr_register()

```
extern int alt_ic_isr_register(alt_u32 ic_id,
 alt_u32 irq,
 alt_isr_func isr,
 void *isr_context,
 void *flags);
```

For comparison, Example 3 shows the function prototype for `alt_irq_register()`, which registers an ISR in the legacy API.

Example 3. Legacy Function alt_irq_register()

```
extern int alt_irq_register (alt_u32 id,
 void* context,
 alt_isr_func handler);
```

The arguments passed into `alt_ic_isr_register()` are slightly different from those passed into `alt_irq_register()`. Table 1 compares the arguments to the two functions.

Table 1. Arguments to alt_ic_isr_register() versus alt_irq_register()

alt_ic_isr_register() Argument	Purpose	alt_irq_register() Argument
`alt_u32 ic_id`	Unique interrupt controller ID as defined in **system.h**.	—
`alt_u32 irq`	Interrupt request (IRQ) number as defined in **system.h**.	`alt_u32 id`
`alt_isr_func isr`	Interrupt service routine (ISR) function pointer	`handler`
`void* isr_context`	Optional pointer to a component-specific data structure.	`context`
`void* flags`	Reserved. Other EIC implementations might use this argument.	None

There are other significant differences between the legacy interrupt API and the enhanced interrupt API. Some of these differences impact the ISR body itself. Notably, the two APIs employ completely different interrupt preemption models. The example code accompanying this application note illustrates many of the differences.

For further information about the other functions in the HAL interrupt APIs, refer to the *Exception Handling* and *HAL API Reference* chapters of the *Nios II Software Developer's Handbook*, and to the *Vectored Interrupt Controller Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

Example Designs

This section provides a brief description of the example designs provided with this application note to demonstrate the usage of the VIC. Additionally, this section provides instructions for running the software examples on the Nios II Embedded Evaluation Kit (NEEK) hardware.

Example Description

The example designs are provided in a file called **AN595_VIC_collateral.zip**.

AN595_VIC_collateral.zip is available on the Literature: Nios II Processor page of the Altera website. A link to the file appears next to *AN595: Vectored Interrupt Controller Usage and Applications* (this document).

Table 2 describes each example design found in the file.

Table 2. Example Designs in AN595_VIC_collateral.zip

Example Name	Folder Name	Description
VIC Basic	**VIC_Example**	A single VIC
VIC Daisy-Chain	**VIC_DaisyChain_Example**	Two daisy-chained VICs
VIC Table-Resident	**VIC_ISRnVectorTable_Example**	VIC with ISR located in vector table
IIC	**VIC_noVIC_Example**	IIC example, for comparison with the VIC examples

The top-level folder in **AN595_VIC_collateral.zip**, called **AN595_VIC_collateral**, contains the following files:

- **run_sw.sh**—Shell script to run one, several or all of the examples
- **README.txt**—Describes the .zip file contents

The hardware for each example is based on the designs shown in Figure 7 and Figure 8. Figure 7 shows the VIC Basic example system in SOPC Builder.

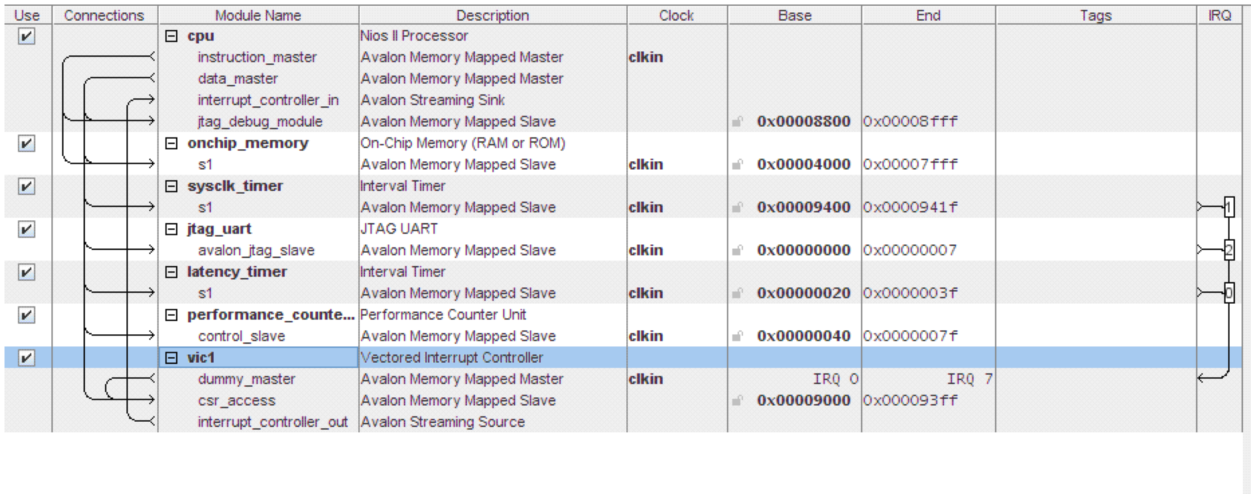
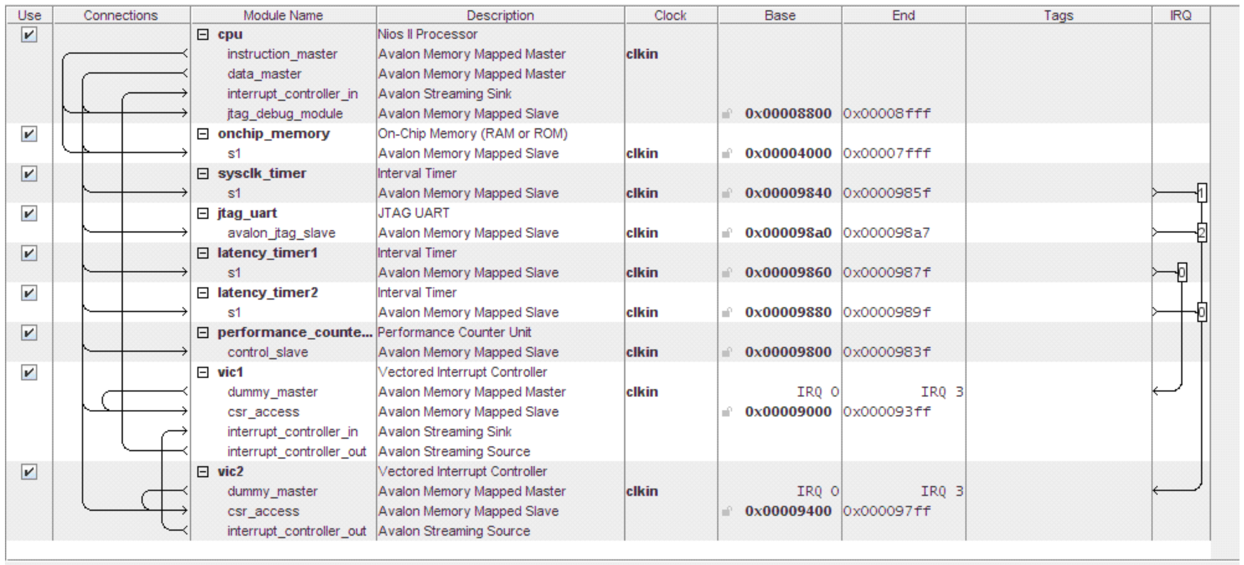


Figure 7. VIC Basic Example

Use	Connections	Module Name	Description	Clock	Base	End	Tags	IRQ
✓		**cpu**	Nios II Processor					
		instruction_master	Avalon Memory Mapped Master	**clkin**				
		data_master	Avalon Memory Mapped Master					
		interrupt_controller_in	Avalon Streaming Sink					
		jtag_debug_module	Avalon Memory Mapped Slave		**0x00008800**	0x00008fff		
✓		**onchip_memory**	On-Chip Memory (RAM or ROM)					
		s1	Avalon Memory Mapped Slave	**clkin**	**0x00004000**	0x00007fff		
✓		**sysclk_timer**	Interval Timer					
		s1	Avalon Memory Mapped Slave	**clkin**	**0x00009400**	0x0000941f		1
✓		**jtag_uart**	JTAG UART					
		avalon_jtag_slave	Avalon Memory Mapped Slave	**clkin**	**0x00000000**	0x00000007		2
✓		**latency_timer**	Interval Timer					
		s1	Avalon Memory Mapped Slave	**clkin**	**0x00000020**	0x0000003f		0
✓		**performance_counte...**	Performance Counter Unit					
		control_slave	Avalon Memory Mapped Slave	**clkin**	**0x00000040**	0x0000007f		
✓		**vic1**	Vectored Interrupt Controller					
		dummy_master	Avalon Memory Mapped Master	**clkin**	IRQ 0	IRQ 7		
		csr_access	Avalon Memory Mapped Slave		**0x00009000**	0x000093ff		
		interrupt_controller_out	Avalon Streaming Source					

Figure 8 shows the VIC Daisy-Chain example system in SOPC Builder.

Figure 8. VIC Daisy-Chain Example



Use	Connections	Module Name	Description	Clock	Base	End	Tags	IRQ
✓		⊟ **cpu**	Nios II Processor					
		instruction_master	Avalon Memory Mapped Master	**clkin**				
		data_master	Avalon Memory Mapped Master					
		interrupt_controller_in	Avalon Streaming Sink					
		jtag_debug_module	Avalon Memory Mapped Slave		**0x00008800**	0x00008fff		
✓		⊟ **onchip_memory**	On-Chip Memory (RAM or ROM)					
		s1	Avalon Memory Mapped Slave	**clkin**	**0x00004000**	0x00007fff		
✓		⊟ **sysclk_timer**	Interval Timer					
		s1	Avalon Memory Mapped Slave	**clkin**	**0x00009840**	0x0000985f		1
✓		⊟ **jtag_uart**	JTAG UART					
		avalon_jtag_slave	Avalon Memory Mapped Slave	**clkin**	**0x000098a0**	0x000098a7		2
✓		⊟ **latency_timer1**	Interval Timer					
		s1	Avalon Memory Mapped Slave	**clkin**	**0x00009860**	0x0000987f		0
✓		⊟ **latency_timer2**	Interval Timer					
		s1	Avalon Memory Mapped Slave	**clkin**	**0x00009880**	0x0000989f		0
✓		⊟ **performance_counte...**	Performance Counter Unit					
		control_slave	Avalon Memory Mapped Slave	**clkin**	**0x00009800**	0x0000983f		
✓		⊟ **vic1**	Vectored Interrupt Controller					
		dummy_master	Avalon Memory Mapped Master	**clkin**	IRQ 0	IRQ 3		
		csr_access	Avalon Memory Mapped Slave		**0x00009000**	0x000093ff		
		interrupt_controller_in	Avalon Streaming Sink					
		interrupt_controller_out	Avalon Streaming Source					
✓		⊟ **vic2**	Vectored Interrupt Controller					
		dummy_master	Avalon Memory Mapped Master	**clkin**	IRQ 0	IRQ 3		
		csr_access	Avalon Memory Mapped Slave		**0x00009400**	0x000097ff		
		interrupt_controller_out	Avalon Streaming Source					

The IIC design is the same as the VIC Basic design, with the VIC and the EIC interface replaced by the IIC. The VIC Table-Resident design is identical to the VIC Basic design.

In each example, the software uses timers in conjunction with performance counters to measure the interrupt performance. Each example's software calculates the performance and sends the results to `stdout`.

AN595_VIC_collateral.zip includes a script, **run_sw.sh**, to run one, several, or all of the example. **run_sw.sh** downloads the SRAM Object File (**.sof**) and the Executable and Linkable Format File (**.elf**) for each example, and executes the code on the NEEK hardware, for the examples that you specify on the command line.

run_sw.sh assumes that you have only one JTAG download cable connected to your host computer. If you have multiple JTAG cables, you must modify **run_sw.sh** to specify the cable connected to your NEEK hardware.

Example Usage

Initially, Altera recommends that you run each example design as distributed, to see the example's performance on your own hardware. Thereafter, you can modify any of the examples to investigate the VIC's performance options, or customize the code for you application.

Execute the following steps to run each example design:

1. Power up your NEEK hardware.
2. Connect the USB cable.

3. Unzip the **AN595_VIC_collateral.zip** file to a working directory, expanding folder names.

   ☞ The path name to your working directory must not contain any spaces.

4. In a Nios II Command Shell, change to the top-level directory, **AN595_VIC_collateral**.
5. At the command prompt, type the following command:

   ```
   ./run_sw.sh↵
   ```

   The script shows a list of options.
6. Run **run_sw.sh** again, using a command-line option that specifies the example you would like to run, or to run all of the examples. Example 4 shows a sample session.

   The **run_sw.sh** script performs the following steps:

   a. Parses the command line argument(s) to determine which example(s) to run

   b. Downloads the **.sof** for the selected example

   c. Downloads the **.elf** for the selected example

   d. Starts **nios2-terminal** to capture the software's output

Software Description

The software for the various example designs is very similar. For example, the difference between the software for the VIC Basic example and the software for the IIC example is the `printf()` call that generates the output to the terminal.

All of the software performs the following steps:

1. Configures the timer used for measurement purposes
2. Registers an interrupt service routine (ISR)
3. Sets a global variable to `0xfeedface`
4. Starts the performance counter to measure the interrupt time
5. Waits for the ISR to set the global variable to `0xfacefeed`
6. Stops the performance counter and computes the interrupt time

The VIC Daisy-Chain example performs the measurement for both VICs connected in the daisy chain, shown in Figure 8 on page 10.

For details about how the VIC Table-Resident example code works, refer to "Positioning the ISR in the Vector Table". For details about performance counter usage in the example software, refer to "Latency Measurement with the Performance Counter" on page 18.

Example 4.

```
[NiosII EDS]$ ./run_sw.sh --VIC_Example

Running software...

Running for VIC_Example
/cygdrive/c/Data/workdir/AN595_VIC_collateral/VIC_Example/software_examples/app/
vic_test /cygdrive/c/Data/workdir/AN595_VIC_collateral
Searching for SOF file:
in ../../../
  VIC_Example.sof

Info: *******************************************************************
Info: Running Quartus II Programmer
Info: Command: quartus_pgm --no_banner --mode=jtag -o p;c:/Data/workdir/
AN595_VIC_collateral/VIC_Example/
VIC_Example.sof
Info: Using programming cable "USB-Blaster [USB-0]"
Info: Started Programmer operation at Mon Nov 2 13:00:33 2009
Info: Configuring device index 1
Info: Device 1 contains JTAG ID code 0x020F30DD
Info: Configuration succeeded -- 1 device(s) configured
Info: Successfully performed operation(s)
Info: Ended Programmer operation at Mon Nov 2 13:00:35 2009
Info: Quartus II Programmer was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 61 megabytes
    Info: Processing ended: Mon Nov 2 13:00:35 2009
    Info: Elapsed time: 00:00:02
    Info: Total CPU time (on all processors): 00:00:00
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00
Pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 4KB in 0.0s
Verified OK
Starting processor at address 0x00004020
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)

Starting VIC Example roundtrip performance test.

Interrupt Time:       48 clocks.

Sending EOT to force an exit.

nios2-terminal: exiting due to ^D on remote

Done...
```

Advanced Topics

This section presents several topics that are useful for advanced interrupt handling.

Positioning the ISR in the Vector Table

If have a critical ISR of small size, you can achieve the best performance by positioning the ISR code directly in the vector table. In this way, you eliminate the overhead of branching from the vector table through the HAL funnel to your ISR.

This section describes how to modify the VIC Basic example software to create the VIC Table-Resident example. Use this example to ensure that you understand the steps. Then you can make the equivalent changes in your custom code.

☞ Positioning an ISR in a vector table is an advanced and error-prone technique, not directly supported by the HAL. You must exercise great caution to ensure that the ISR code fits in the vector table entry. If your ISR overflows the vector table entry, it corrupts other entries in the vector table, and your entire interrupt handling system.

When locate your ISR in the vector table, it does not need to be registered. Do not call `alt_ic_isr_register()`, because it overwrites the contents of the vector table.

When the ISR is in the vector table, the HAL does not provide funnel code. Therefore, the ISR code must perform any context-switching actions normally handled by the funnel. Funnel context switching can include some or all of the following actions:

- Saving and restoring registers
- Managing preemption
- Managing the stack pointer

To create the fastest possible ISR, minimize or eliminate the context-switching actions your ISR must perform by conforming to the following guidelines:

- Write the ISR in assembly language.
- Assign a shadow register set for the ISR's use.
- Ensure that the ISR cannot be preempted by another ISR using the same register set. By default, preemption within a register set is disabled on the Nios II processor. You can also ensure this condition by giving the ISR exclusive access to its register set.

☞ The VIC Table-Resident example requires modifying a BSP-generated file, **altera_vic1_vector_tbl.S**. If you regenerate the BSP after making these modifications, the Nios II Software Build Tools regenerate **altera_vic1_vector_tbl.S**, and your changes are overwritten.

Increase the Vector Table Entry Size

To insert the ISR in the vector table, you must increase the size of the vector entries so that your entire ISR fits in a vector table entry. Use the `altera_vic_driver.<vic_instance>.vec_size` BSP setting to adjust the vector table entry size. On the Nios II Software Build Tools command line, you can manipulate this setting with the `--set` command-line option. You can also modify this setting in the Nios II BSP Editor.

In the VIC Table-Resident example, <*vic_instance*> is `VIC1` and <*size*> is set to 256 bytes.

Do Not Register the ISR

Remove the call to `alt_ic_isr_register()` for the interrupt that you place in the vector table. Replace it with an `alt_ic_irq_enable()` call. You must not call `alt_ic_isr_register()`, because it overwrites the contents of the vector table, destroying the body of your ISR.

Insert ISR in Vector Table

In the VIC Table-Resident example included with this document, the ISR code is in a file called **vector.h** in the BSP folder.

To insert this code in the vector table, execute the following steps:

1. Generate the BSP by running the **create-this-bsp** script.
2. Modify **altera_vic1_vector_tbl.S** as shown in Example 4.

Example 5. Modifications to altera_vic1_vector_tbl.S

```
#include "altera_vic_funnel.h"
#include "vector.h"            /* ADD THIS LINE MANUALLY */
    .section .text
    .align 2
    .globl VIC1_VECTOR_TABLE
VIC1_VECTOR_TABLE:
    MY_ISR 256                 /* THIS LINE REPLACES THE FIRST VECTOR TABLE ENTRY */
    ALT_SHADOW_NON_PREEMPTIVE_INTERRUPT 256
    ALT_SHADOW_NON_PREEMPTIVE_INTERRUPT 256
    ALT_SHADOW_NON_PREEMPTIVE_INTERRUPT 256
    ALT_SHADOW_NON_PREEMPTIVE_INTERRUPT 256
```

After completion of these steps, build the software, run it, and observe the reported interrupt time. This example is about 20 clock cycles faster than the unmodified VIC Basic example.

Some variation is likely for reasons discussed in “Real-Time Latency Concerns”.

Refer to the example code for further details.

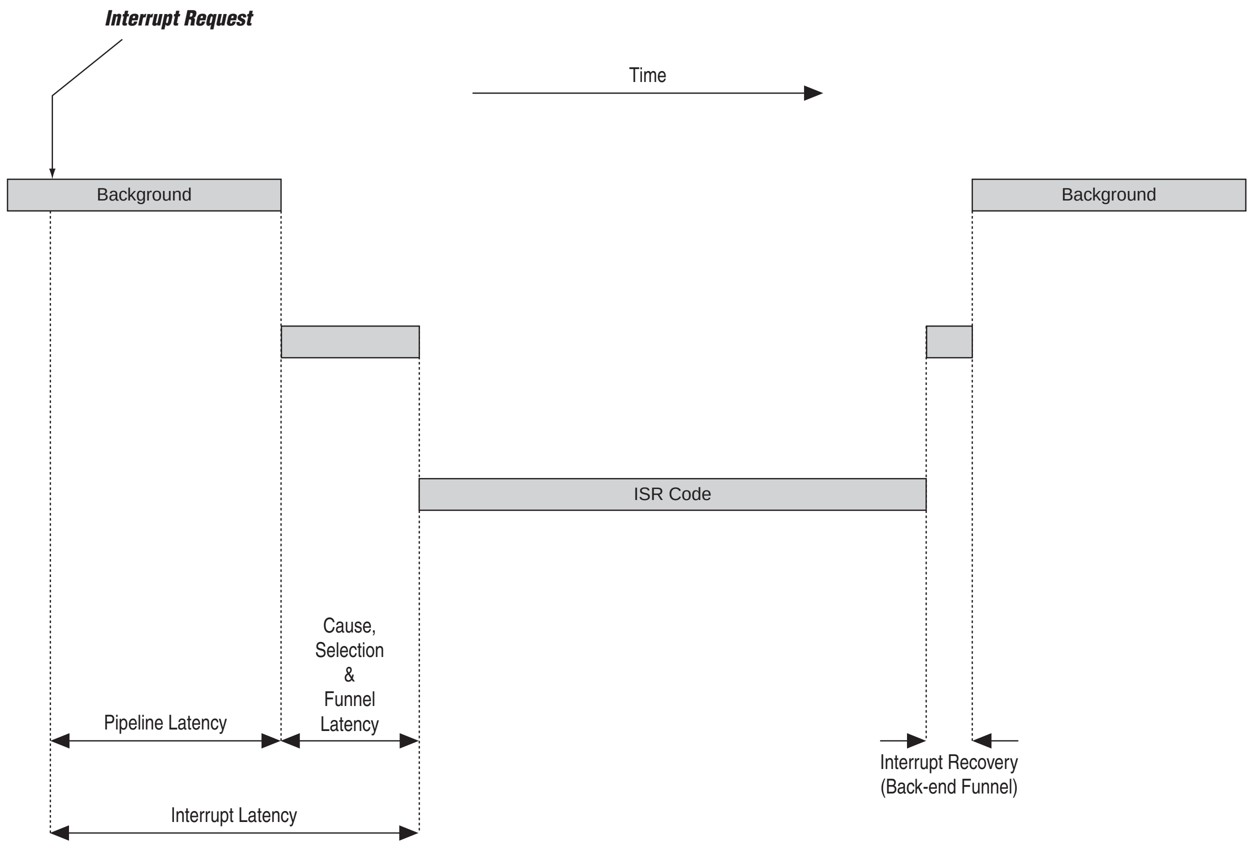
Real-Time Latency Concerns

This section presents an overview of interrupt latency, the elements that combine to determine interrupt latency, and methods for measuring it. The following elements comprise interrupt latency:

- Pipeline Latency—“Pipeline Latency”
- Cause Latency—Described in “Cause Latency”
- Selection Latency—Described in “Selection Latency”
- Funnel Latency—Described in “Funnel Latency”
- Compiler-related latency—Described in “Compiler-Related Latency” on page 18

The interrupt latency diagram in Figure 9 illustrates these elements.

Figure 9. The Elements of Interrupt Latency



This section summarizes each element of latency and describes how to measure latency. The accompanying example designs use the performance counter core to capture all of the timing measurements. Performance counter core usage is described in "Latency Measurement with the Performance Counter" on page 18.

For information about the performance counter core, refer to the *Performance Counter Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook.*

Pipeline Latency

Pipeline latency is defined as the number of clock cycles between an interrupt signal being asserted and the execution of the first instruction at the exception vector. It can vary widely, depending on the type of memory the processor is executing from and the impact of other master ports in your hardware. Theoretically, this time could be infinite if an ill-behaved master port blocks the processor from accessing memory, freezing the processor.

Cause Latency

Cause latency is the time required for the processor to identify an exception as a hardware interrupt. With an EIC, such as the VIC, the cause latency is zero because each hardware interrupt has a dedicated interrupt vector address, separate from the software exception vector address.

Selection Latency

Selection latency is the time required for the system to transfer control to the correct interrupt vector, depending on which interrupt is triggered. The selection latency with the VIC component depends on the number of interrupts that it services. Table 3 outlines selection latency on a single VIC as a function of the number of interrupts.

Table 3. The Components of VIC Latency

Total Number of Interrupts	Interrupt Request Clock Delay (clocks)	Priority Processing Clock Delay (clocks)	Vector Generation Clock Delay (clocks)	Total Interrupt Latency (clocks)
1	2	0	1	3
2—4	2	1	1	4
5—16	2	2	1	5
17—32	2	3	1	6

Funnel Latency

Funnel latency is the time required for the interrupt funnel to switch context. Funnel latency can include saving and restoring registers, managing preemption, and managing the stack pointer. Funnel latency depends on the following factors:

- Whether a separate interrupt stack is used
- The number of clock cycles required for load and store instructions
- Whether the interrupt requires switching to a different register set
- Whether the interrupt is preempting another interrupt within the same register set
- Whether preemption within the register set is allowed

Preemption within the register set requires special attention. The HAL VIC driver provides special funnel code if an interrupt is allowed to preempt another interrupt assigned to the same register set. In this case, the funnel incurs additional overhead to save and restore the register contents. When creating the BSP, you can control preemption within the register set by using the VIC driver's `altera_vic_driver_enable_preemption_rs_<n>` setting.

☞ For information about the `altera_vic_driver_enable_preemption_rs_<n>` setting, refer to "Altera HAL Software Programming Model" in the *Vectored Interrupt Controller Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook.*

Table 4 and Table 5 show the funnel latencies for various configurations.

Table 4. Single Stack HAL latency

Funnel Type	Clock Cycles Required for Load or Store (1)	
	1	2
Shadow register set, preemption within the register set disabled	10	13
Shadow register set, preemption within the register set enabled	42 Same register set (`sstatus.SRS=0`)	64 Same register set (`sstatus.SRS=0`)
	26 Different register set (`sstatus.SRS=1`)	32 Different register set (`sstatus.SRS=1`)

Note to Table 4:

(1) With tightly-coupled memory, the Nios II processor can execute a load or store instruction in 1 clock cycle. With onchip memory, not tightly-coupled, the processor requires two clock cycles.

Table 5. Separate Interrupt Stack HAL Latency

Funnel Type	Clock Cycles Required for Load or Store (1)	
	1	2
Shadow register set, preemption within the register set disabled	11 Not preempting another interrupt (`sstatus.IH=0`)	14 Not preempting another interrupt (`sstatus.IH=0`)
	12 Preempting another interrupt (`sstatus.IH=1`)	15 Preempting another interrupt (`sstatus.IH=1`)
Shadow register set, preemption within the register set enabled	42 Same register set (`sstatus.SRS=0`)	64 Same register set (`sstatus.SRS=0`)
	27 ■ Different register set (`sstatus.SRS=1`) ■ Not preempting another interrupt (`sstatus.IH=0`)	33 ■ Different register set (`sstatus.SRS=1`) ■ Not preempting another interrupt (`sstatus.IH=0`)
	28 ■ Different register set (`sstatus.SRS=1`) ■ Preempting another interrupt (`sstatus.IH=1`)	34 ■ Different register set (`sstatus.SRS=1`) ■ Preempting another interrupt (`sstatus.IH=1`)

Note to Table 5:

(1) With tightly-coupled memory, the Nios II processor can execute a load or store instruction in 1 clock cycle. With onchip memory, not tightly-coupled, the processor requires two clock cycles.

In Table 4 and Table 5, notice that the lowest latencies occur under the following conditions:

- A different register set—Shadow register set switch; the ISR runs in a different register set from the interrupted task, eliminating any need to save or restore registers.
- Preemption (nesting) within the register set disabled.

Conversely, the highest latencies occur under the following conditions:

- The same register set—No shadow register set switch; the ISR runs in the same register set as the interrupted task, requiring the funnel code to save and restore registers.
- Preemption within the register set enabled.

Of these two important factors, preemption makes the largest difference in latencies. With preemption disabled, much lower latencies occur regardless of other factors.

Compiler-Related Latency

The GNU C compiler creates a prologue and epilogue for many C functions, including ISRs. The prologue and epilogue are code sequences that take care of housekeeping tasks, such as saving and restoring context for the C runtime environment. The time required for the prologue and epilogue is called compiler-related latency.

The C compiler generates a prologue and epilogue as needed. If compiler optimization is enabled, and the routine is compact, with few local variables, the prologue and epilogue are usually omitted. You can determine whether a prologue and epilogue are generated by examining the function's assembly code.

Compiler latency normally has only a minor impact on overall interrupt servicing performance. If you are concerned about compiler latency, you have two options:

- Enable compiler optimizations, and simplify your ISR, minimizing local variables.
- Write your ISR in assembly language.

Latency Measurement with the Performance Counter

The Altera Complete Design Suite provides tools that enable you to make fast, accurate performance measurements. All examples included with this document use the Performance Counter component to measure interrupt latency.

The examples execute the following steps to measure the total time spent to service an interrupt:

1. Initialize a global variable, `interrupt_watch_value`, to a known value, `0xfeedface`.
2. Set up a timer interrupt, registering an ISR that sets `interrupt_watch_value` to `0xfacefeed`.
3. Start the timer.
4. Wait in a `while()` loop until `interrupt_watch_value` becomes `0xfacefeed`.
5. Immediately after exiting the `while()` loop, stop the performance counter, compute clock cycles and display the calculated value on `stdout`.

You can use similar methods to determine the real-time interrupt latencies in your system.

Using Software Interrupts

Software can trigger any VIC interrupt by writing to the appropriate VIC control and status register (CSR). Software can trigger the interrupt connected to any hardware interrupt source, as well as interrupts that are not connected to hardware (software-only interrupts).

Triggering an interrupt from software is useful for debugging. Software can control exactly when an interrupt is triggered, and measure the system’s interrupt response.

You can use a software-only interrupt to reprioritize an interrupt. An ISR that responds to a high-priority hardware interrupt can perform the minimum processing required by the hardware, and then trigger a software-only interrupt at a lower priority level to complete the interrupt processing.

The following functions are available for managing software interrupts:

- `alt_vic_sw_interrupt_set()`
- `alt_vic_sw_interrupt_clear()`
- `alt_vic_sw_interrupt_status()`

The implementations of these functions are in **bsp/hal/drivers/src/altera_vic_sw_intr.c** after you generate the BSP. For detailed descriptions of the functions, refer to the *Vectored Interrupt Controller Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

Example 6 shows how to register a software interrupt.

☞ You must define a value for the interrupt number in `SOFT_IRQ`.

Example 6. Registering a Software Interrupt

```
alt_ic_isr_register(
    VIC1_INTERRUPT_CONTROLLER_ID,
    SOFT_IRQ,
    soft_interrupt_latency_irq,
    NULL, NULL)
```

For comparison purposes, Example 7 shows timer interrupt registration.

Example 7. Registering a Timer Interrupt (for Comparison)

```
alt_ic_isr_register(
    LATENCY_TIMER_IRQ_INTERRUPT_CONTROLLER_ID,
    LATENCY_TIMER_IRQ,
    timer_interrupt_latency_irq,
    LATENCY_TIMER_BASE,
    NULL);
```

The following code generates a software interrupt:

```
alt_vic_sw_interrupt_set(VIC1_INTERRUPT_CONTROLLER_ID, SOFT_IRQ);
```

Conclusion

This document describes the benefits and usage of the VIC and the Nios II EIC interface. The hardware and software descriptions and usage examples provide you with enough information to get started using the VIC in your system. The advanced topic coverage provides methods to increase the performance gains that you can achieve with the VIC.

Referenced Documents

This application note references the following documents:

- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*
- *Programming Model* chapter of the *Nios II Processor Reference Handbook*
- *Performance Counter Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*
- *Volume 4: SOPC Builder* in the *Quartus II Handbook*
- *Vectored Interrupt Controller Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*
- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*
- *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*
- Nios Community Wiki (www.nioswiki.com)

Document Revision History

Table 6 shows the revision history for this document.

Table 6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2009 v1.0	Initial Release.	—