8. Exception Handling

NII52006-7.2.0

Introduction

This chapter discusses how to write programs to handle exceptions in the Nios® II processor architecture. Emphasis is placed on how to process hardware interrupt requests by registering a user-defined interrupt service routine (ISR) with the hardware abstraction layer (HAL).

This chapter contains the following sections:

For low-level details of handling exceptions and interrupts on the Nios II architecture, see the *Programming Model* chapter of the *Nios II Processor Reference Handbook.*

Nios II Exceptions Overview

Nios II exception handling is implemented in classic RISC fashion, i.e., all exception types are handled by a single exception handler. As such, all exceptions (hardware and software) are handled by code residing at a single location called the "exception address".

The Nios II processor provides the following exception types:

- Hardware interrupts
- Software exceptions, which fall into the following categories:
  - Unimplemented instructions
  - Software traps
  - Other exceptions

Exception Handling Concepts

The following list outlines basic exception handling concepts, with the HAL terms used for each one:

- **application context** — the status of the Nios II processor and the HAL during normal program execution, outside of the exception handler.
- **context switch** — the process of saving the Nios II processor's registers on an exception, and restoring them on return from the interrupt service routine.
- **exception** — any condition or signal that interrupts normal program execution.
- **exception handler** — the complete system of software routines, which service all exceptions and pass control to ISRs as necessary.
- **exception overhead** — additional processing required by exception processing. The exception overhead for a program is the sum of all the time occupied by all context switches.
- **hardware interrupt** — an exception caused by a signal from a hardware device.
- **implementation-dependent instruction** — a Nios II processor instruction that is not supported on all implementations of the Nios II core. For example, the `mul` and `div` instructions are implementation-dependent, because they are not supported on the Nios II/e core.
- **interrupt context** — the status of the Nios II processor and the HAL when the exception handler is executing.
- **interrupt request (IRQ)** — a signal from a peripheral requesting a hardware interrupt.
- **interrupt service routine (ISR)** — a software routine that handles an individual hardware interrupt.
- **invalid instruction** — an instruction that is not defined for any implementation of the Nios II processor.
- **software exception** — an exception caused by a software condition. This includes unimplemented instructions and `trap` instructions.
- **unimplemented instruction** — an implementation-dependent instruction that is not supported on the particular Nios II core implementation that is in your system. For example, in the Nios II/e core, `mul` and `div` are unimplemented.

- **other exception** — an exception which is not a hardware interrupt nor a trap.

How the Hardware Works

The Nios II processor can respond to software exceptions and hardware interrupts. Thirty-two independent hardware interrupt signals are available. These interrupt signals allow software to prioritize interrupts, although the interrupt signals themselves have no inherent priority.

When the Nios II processor responds to an exception, it does the following things:

1. Saves the `status` register in `estatus`. This means that if hardware interrupts are enabled, the `EPIE` bit of `estatus` is set.
2. Disables hardware interrupts.
3. Saves the next execution address in `ea` (`r29`).
4. Transfers control to the Nios II processor exception address.

☞ Nios II exceptions and interrupts are not vectored. Therefore, the same exception address receives control for all types of interrupts and exceptions. The exception handler at that address must determine the type of exception or interrupt.

For details about the Nios II processor exception and interrupt controller, see the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

ISRs

Software often communicates with peripheral devices using interrupts. When a peripheral asserts its IRQ, it causes an exception to the processor's normal execution flow. When such an IRQ occurs, an appropriate ISR must handle this interrupt and return the processor to its pre-interrupt state upon completion.

When you use the Nios II IDE to create a system library project, the IDE includes all needed ISRs. You do not need to write HAL ISRs unless you are interfacing to a custom peripheral. For reference purposes, this section describes the framework provided by the HAL system library for handling hardware interrupts.

You can also look at existing handlers for Altera® SOPC Builder components for examples of how to write HAL ISRs.

For more details about the Altera-provided HAL handlers, see the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook.*

HAL API for ISRs

The HAL system library provides an API to help ease the creation and maintenance of ISRs. This API also applies to programs based on a real-time operating system (RTOS) such as MicroC/OS-II, because the full HAL API is available to RTOS-based programs. The HAL API defines the following functions to manage hardware interrupt processing:

- `alt_irq_register()`
- `alt_irq_disable()`
- `alt_irq_enable()`
- `alt_irq_disable_all()`
- `alt_irq_enable_all()`
- `alt_irq_interruptible()`
- `alt_irq_non_interruptible()`
- `alt_irq_enabled()`

For details on these functions, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Using the HAL API to implement ISRs entails the following steps:

1. Write your ISR that handles interrupts for a specific device.

2. Your program must register the ISR with the HAL by calling the `alt_irq_register()` function. `alt_irq_register()` enables interrupts for you, by calling `alt_irq_enable_all()`.

Writing an ISR

The ISR you write must match the prototype that `alt_irq_register()` expects to see. The prototype for your ISR function must match the prototype:

```
void isr (void* context, alt_u32 id)
```

The parameter definitions of `context` and `id` are the same as for the `alt_irq_register()` function.

From the point of view of the HAL exception handling system, the most important function of an ISR is to clear the associated peripheral's interrupt condition. The procedure for clearing an interrupt condition is specific to the peripheral.

For details, see the relevant chapter in the *Quartus® II Handbook, Volume 5: Altera Embedded Peripherals*.

When the ISR has finished servicing the interrupt, it must return to the HAL exception handler.

If you write your ISR in assembly language, use `ret` to return. The HAL exception handler issues an `eret` after restoring the application context.

Restricted Environment

ISRs run in a restricted environment. A large number of the HAL API calls are not available from ISRs. For example, accesses to the HAL file system are not permitted. As a general rule, when writing your own ISR, never include function calls that can block waiting for an interrupt.

The *HAL API Reference* chapter of the *Nios II Software Developer's Handbook* identifies those API functions that are not available to ISRs.

Be careful when calling ANSI C standard library functions inside of an ISR. Avoid using the C standard library I/O API, because calling these functions can result in deadlock within the system, i.e., the system can become permanently blocked within the ISR.

In particular, do not call `printf()` from within an ISR unless you are certain that `stdout` is mapped to a non-interrupt-based device driver. Otherwise, `printf()` can deadlock the system, waiting for an interrupt that never occurs because interrupts are disabled.

Registering an ISR

Before the software can use an ISR, you must register it by calling `alt_irq_register()`. The prototype for `alt_irq_register()` is:

```
int alt_irq_register (alt_u32 id,
                      void*   context,
                      void    (*isr)(void*, alt_u32));
```

The prototype has the following parameters:

- `id` is the hardware interrupt number for the device, as defined in **system.h**. Interrupt priority corresponds inversely to the IRQ number. Therefore, IRQ_0 represents the highest priority interrupt and IRQ_{31} is the lowest.
- `context` is a pointer used to pass context-specific information to the ISR, and can point to any ISR-specific information. The context value is opaque to the HAL; it is provided entirely for the benefit of the user-defined ISR.
- `isr` is a pointer to the function that is called in response to IRQ number id. The two input arguments provided to this function are the `context` pointer and `id`. Registering a null pointer for `isr` results in the interrupt being disabled.

The HAL registers the ISR by the storing the function pointer, `isr`, in a lookup table. The return code from `alt_irq_register()` is zero if the function succeeded, and nonzero if it failed.

If the HAL registers your ISR successfully, the associated Nios II interrupt (as defined by `id`) is enabled on return from `alt_irq_register()`.

Hardware-specific initialization might also be required.

When a specific IRQ occurs, the HAL looks up the IRQ in the lookup table and dispatches the registered ISR.

For details of interrupt initialization specific to your peripheral, see the relevant chapter in the *Quartus II Handbook, Volume 5: Altera Embedded Peripherals*. For details on `alt_irq_register()`, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Enabling and Disabling ISRs

The HAL provides the functions `alt_irq_disable()`, `alt_irq_enable()`, `alt_irq_disable_all()`, `alt_irq_enable_all()`, and `alt_irq_enabled()` to allow a program to disable interrupts for certain sections of code, and re-enable them later. `alt_irq_disable()` and `alt_irq_enable()` allow you to disable and enable individual interrupts. `alt_irq_disable_all()` disables all interrupts, and returns a context value. To re-enable interrupts, you call `alt_irq_enable_all()` and pass in the context parameter. In this way, interrupts are returned to their state prior to the call to `alt_irq_disable_all()`. `alt_irq_enabled()` returns non-zero if interrupts are enabled, allowing a program to check on the status of interrupts.

☞ Disable interrupts for as short a time as possible. Maximum interrupt latency increases with the amount of time interrupts are disabled. For more information about disabled interrupts, see "Keep Interrupts Enabled" on page 8–11.

For details on these functions, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

C Example

The following code illustrates an ISR that services an interrupt from a button PIO. This example is based on a Nios II system with a 4-bit PIO peripheral connected to push-buttons. An IRQ is generated any time a button is pushed. The ISR code reads the PIO peripheral's edge-capture register and stores the value to a global variable. The address of the global variable is passed to the ISR via the context pointer.

Example: An ISR to Service a Button PIO IRQ

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

static void handle_button_interrupts(void* context, alt_u32 id)
{
  /* cast the context pointer to an integer pointer. */
  volatile int* edge_capture_ptr = (volatile int*) context;

  /*
   * Read the edge capture register on the button PIO.
   * Store value.
   */
  *edge_capture_ptr =
    IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);

  /* Write to the edge capture register to reset it. */
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);

  /* reset interrupt capability for the Button PIO. */
  IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
}
```

The following code shows an example of the code for the main program that registers the ISR with the HAL.

Example: Registering the Button PIO ISR with the HAL

```
#include "sys/alt_irq.h"
#include "system.h"

...
/* Declare a global variable to hold the edge capture value. */
volatile int edge_capture;
...
```

```
/* Initialize the button_pio. */
static void init_button_pio()
{
    /* Recast the edge_capture pointer to match the
        alt_irq_register() function prototype. */
    void* edge_capture_ptr = (void*) &edge_capture;

    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);

    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);

    /* Register the ISR. */
    alt_irq_register( BUTTON_PIO_IRQ,
                        edge_capture_ptr,
                        handle_button_interrupts );
}
```

Based on this code, the following execution flow is possible:

1. Button is pressed, generating an IRQ.

2. The HAL exception handler is invoked and dispatches the `handle_button_interrupts()` ISR.

3. `handle_button_interrupts()` services the interrupt and returns.

4. Normal program operation continues with an updated value of `edge_capture`.

Further software examples that demonstrate implementing ISRs are installed with the Nios II Embedded Design Suite (EDS), such as the `count_binary` example project template.

ISR Performance Data

This section provides performance data related to ISR processing on the Nios II processor. The following three key metrics determine ISR performance:

- Interrupt latency—the time from when an interrupt is first generated to when the processor runs the first instruction at the exception address.
- Interrupt response time—the time from when an interrupt is first generated to when the processor runs the first instruction in the ISR.
- Interrupt recovery time—the time taken from the last instruction in the ISR to return to normal processing.

Because the Nios II processor is highly configurable, there is no single typical number for each metric. This section provides data points for each of the Nios II cores under the following assumptions:

- All code and data is stored in on-chip memory.
- The ISR code does not reside in the instruction cache.
- The software under test is based on the Altera-provided HAL exception handler system.
- The code is compiled using compiler optimization level "–O3", or high optimization.

Table 8–1 lists the interrupt latency, response time, and recovery time for each Nios II core.

Table 8–1. Interrupt Performance Data (1)

Core	Latency	Response Time	Recovery Time
Nios II/f	10	105	62
Nios II/s	10	128	130
Nios II/e	15	485	222

Note to Table 8–1:
(1) The numbers indicate time measured in CPU clock cycles.

The results you experience in a specific application can vary significantly based on several factors discussed in the next section.

Improving ISR Performance

If your software uses interrupts extensively, the performance of ISRs is probably the most critical determinant of your overall software performance. This section discusses both hardware and software strategies to improve ISR performance.

Software Performance Improvements

In improving your ISR performance, you probably consider software changes first. However, in some cases it might require less effort to implement hardware design changes that increase system efficiency. For a discussion of hardware optimizations, see "Hardware Performance Improvements" on page 8–13.

The following sections describe changes you can make in the software design to improve ISR performance.

Move Lengthy Processing to Application Context

ISRs provide rapid, low latency response to changes in the state of hardware. They do the minimum necessary work to clear the interrupt condition and then return. If your ISR performs lengthy, noncritical processing, it interferes with more critical tasks in the system.

If lengthy processing is needed, design your software to perform this processing outside of the interrupt context. The ISR can use a message-passing mechanism to notify the application code to perform the lengthy processing tasks.

Deferring a task is simple in systems based on an RTOS such as MicroC/OS-II. In this case, you can create a thread to handle the processor-intensive operation, and the ISR can communicate with this thread using any of the RTOS communication mechanisms, such as event flags or message queues.

You can emulate this approach in a single-threaded HAL-based system. The main program polls a global variable managed by the ISR to determine whether it needs to perform the processor-intensive operation.

Move Lengthy Processes to Hardware

Processor-intensive tasks must often transfer large amounts of data to and from peripherals. A general-purpose CPU such as the Nios II processor is not the most efficient way to do this.

Use Direct Memory Access (DMA) hardware if it is available.

For information about programming with DMA hardware, refer to the *Using DMA Devices* section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Increase Buffer Size

If you are using DMA to transfer large data buffers, the buffer size can affect performance. Small buffers imply frequent IRQs, which lead to high overhead.

Increase the size of the transaction data buffer(s).

Use Double Buffering

Using DMA to transfer large data buffers might not provide a large performance increase if the Nios II processor must wait for DMA transactions to complete before it can perform the next task.

Double buffering allows the Nios II processor to process one data buffer while the hardware is transferring data to or from another.

Keep Interrupts Enabled

When interrupts are disabled, the Nios II processor cannot respond quickly to hardware events. Buffers and queues can fill or overflow. Even in the absence of overflow, maximum interrupt processing time can increase after interrupts are disabled, because the ISRs must process data backlogs.

Disable interrupts as little as possible, and for the briefest time possible.

Instead of disabling all interrupts, call `alt_irq_disable()` and `alt_irq_enable()` to enable and disable individual IRQs.

To protect shared data structures, use RTOS structures such as semaphores.

Disable all interrupts only during critical system operations. In the code where interrupts are disabled, perform only the bare minimum of critical operations, and re-enable interrupts immediately.

Use Fast Memory

ISR performance depends upon memory speed.

Place the ISRs and the stack in the fastest available memory.

For best performance, place the stack in on-chip memory, preferably tightly-coupled memory, if available.

If it is not possible to place the main stack in fast memory, you can use a private exception stack, mapped to a fast memory section. However, the private exception stack entails some additional context switch overhead, so use it only if you are able to place it in significantly faster memory. You can specify a private exception stack on the **System properties** page of the Nios II IDE.

For more information about mapping memory, see the "Memory Usage" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. For more information on tightly-coupled memory, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Use Nested ISRs

The HAL system library disables interrupts when it dispatches an ISR. This means that only one ISR can execute at any time, and ISRs are executed on a first-come-first-served basis.This reduces the system overhead associated with interrupt processing, and simplifies ISR development, because the ISR does not need to be reentrant.

However, first-come first-served execution means that the HAL interrupt priorities only have effect if two IRQs are asserted on the same application-level instruction cycle. A low-priority interrupt occurring before a higher-priority IRQ can prevent the higher-priority ISR from executing. This is a form of priority inversion, and it can have a significant impact on ISR performance in systems that generate frequent interrupts.

A software system can achieve full interrupt prioritization by using nested ISRs. With nested ISRs, higher priority interrupts are allowed to interrupt lower-priority ISRs.

This technique can improve the interrupt latency of higher priority ISRs.

☞ Nested ISRs increase the processing time for lower priority interrupts.

If your ISR is very short, it might not be worth the overhead to re-enable higher-priority interrupts. Enabling nested interrupts in a short ISR can actually increase the interrupt latency of higher priority interrupts.

☞ If you use a private exception stack, you cannot nest interrupts. For more information about private exception stacks, see "Use Fast Memory" on page 8–11.

To implement nested interrupts, use the `alt_irq_interruptible()` and `alt_irq_non_interruptible()` functions to bracket code within a processor-intensive ISR. After the call to `alt_irq_interruptible()`, higher priority IRQs can interrupt the running ISR. When your ISR calls `alt_irq_non_interruptible()`, interrupts are disabled as they were before `alt_irq_interruptible()`.

☞ If your ISR calls `alt_irq_interruptible()`, it must call `alt_irq_non_interruptible()` before returning. Otherwise, the HAL exception handler might lock up.

Use Compiler Optimization

For the best performance both in exception context and application context, use compiler optimization level `-O3`. Level `-O2` also produces good results. Removing optimization altogether significantly increases interrupt response time.

For further information about compiler optimizations, refer to the *Reducing Code Footprint* section in the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Hardware Performance Improvements

There are several simple hardware changes that can provide a substantial improvement in ISR performance. These changes involve editing and regenerating the SOPC Builder module, and recompiling the Quartus II design.

In some cases, these changes also require changes in the software architecture or implementation. For a discussion of these and other software optimizations, see "Software Performance Improvements" on page 8–9.

The following sections describe changes you can make in the hardware design to improve ISR performance.

Add Fast Memory

Increase the amount of fast on-chip memory available for data buffers. Ideally, implement tightly-coupled memory which the software can use for buffers.

For further information about tightly-coupled memory, refer to the *Cache and Tightly-Coupled Memory* chapter in the *Nios II Processor Reference Handbook*, or to the *Using Nios II Tightly Coupled Memory Tutorial*.

Add a DMA Controller

A DMA controller performs bulk data transfers, reading data from a source address range and writing the data to a different address range. Add DMA controllers to move large data buffers. This allows the Nios II processor to carry out other tasks while data buffers are being transferred.

For information about DMA controllers, see the *DMA Controller Core* chapter of the *Quartus II Handbook, Volume 5: Embedded Peripherals*.

Place the Exception Handler Address in Fast Memory

For the fastest execution of exception code, place the exception address in a fast memory device. For example, an on-chip RAM with zero waitstates is preferable to a slow SDRAM. For best performance, store exception handling code and data in tightly-coupled memory. The Nios II EDS includes example designs that demonstrate the use of tightly-coupled memory for ISRs.

Use a Fast Nios II Core

For processing in both the interrupt context and the application context, the Nios II/f core is the fastest, and the Nios II/e core (designed for small size) is the slowest.

Select Interrupt Priorities

When selecting the IRQ for each peripheral, bear in mind that the HAL hardware interrupt handler treats IRQ_0 as the highest priority. Assign each peripheral's interrupt priority based on its need for fast servicing in the overall system. Avoid assigning multiple peripherals to the same IRQ.

Use the Interrupt Vector Custom Instruction

The Nios II processor core offers an interrupt vector custom instruction which accelerates interrupt vector dispatch in the Hardware Abstraction Layer (HAL). You can choose to include this custom instruction to improve your program's interrupt response time.

When the interrupt vector custom instruction is present in the Nios II processor, the HAL source detects it at compile time and generates code using the custom instruction.

For further information about the interrupt vector custom instruction, see the *Interrupt Vector Custom Instruction* section in the chapter entitled *Instantiating the Nios II Processor in SOPC Builder* in the *Nios II Processor Reference Handbook.*

Debugging ISRs

You can debug an ISR with the Nios II IDE by setting breakpoints within the ISR. The debugger completely halts the processor upon reaching a breakpoint. In the meantime, however, the other hardware in your system continues to operate. Therefore, it is inevitable that other IRQs are ignored while the processor is halted. You can use the debugger to step through the ISR code, but the status of other interrupt-driven device drivers is generally invalid by the time you return the processor to normal execution. You have to reset the processor to return the system to a known state.

The `ipending` register (`ctl4`) is masked to all zeros during single stepping. This masking prevents the processor from servicing IRQs that are asserted while you single-step through code. As a result, if you try to single step through a part of the exception handler code (e.g. `alt_irq_entry()` or `alt_irq_handler()`) that reads the `ipending` register, the code does not detect any pending IRQs. This issue does not affect debugging software exceptions. You can set breakpoints within your ISR code (and single step through it), because the exception handler has already used `ipending` to determine which IRQ caused the exception.

Summary of Guidelines for Writing ISRs

This section summarizes guidelines for writing ISRs for the HAL framework:

- Write your ISR function to match the prototype: `void isr (void* context, alt_u32 id)`.
- Register your ISR using the `alt_irq_register()` function provided by the HAL API.
- Do not use the C standard library I/O functions, such as `printf()`, inside of an ISR.

HAL Exception Handler Implementation

This section describes the HAL exception handler implementation. This is one of many possible implementations of an exception handler for the Nios II processor. Some features of the HAL exception handler are constrained by the Nios II hardware, while others are designed to provide generally useful services.

This information is for your reference. You can take advantage of the HAL exception services without a complete understanding of the HAL implementation. For details of how to install ISRs using the HAL application programming interface (API), see "ISRs" on page 8–3.

Exception Handler Structure

The exception handling system consists of the following components:

- The top-level exception handler
- The hardware interrupt handler
- The software exception handler
- An ISR for each peripheral that generates interrupts.

When the Nios II processor generates an exception, the top-level exception handler receives control. The top-level exception handler passes control to either the hardware interrupt handler or the software exception handler. The hardware interrupt handler passes control to one or more ISRs.

Each time an exception occurs, the exception handler services either a software exception or hardware interrupts, with hardware interrupts having a higher priority. The HAL does not support nested exceptions, but can handle multiple hardware interrupts per context switch. For details, see "Hardware Interrupt Handler" on page 8–18.

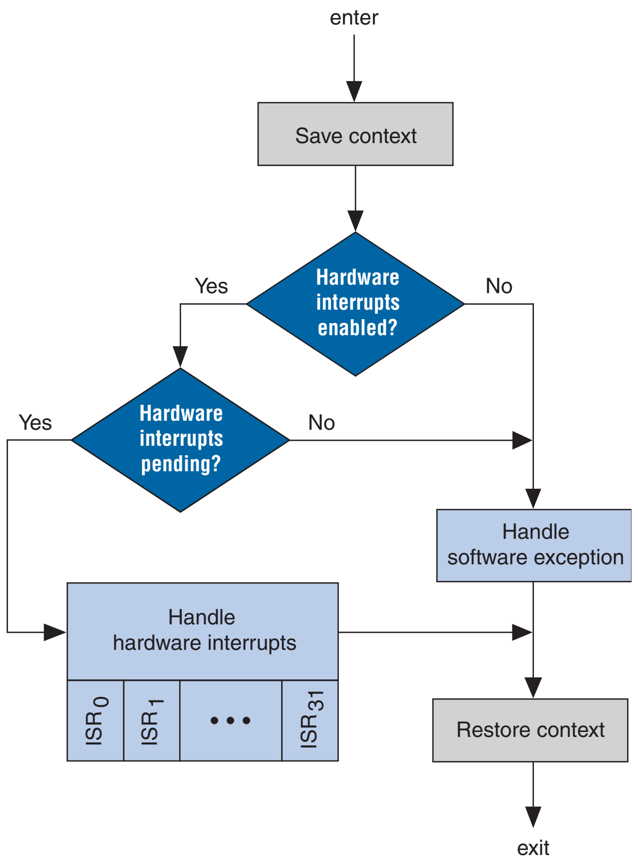
Top-Level Exception Handler

The top-level exception handler provided with the HAL system library is located at the Nios II processor's exception address. When an exception occurs and control transfers to the exception handler, it does the following:

1. Creates the private exception stack (if specified)
2. Stores register values onto the stack
3. Determines the type of exception, and passes control to the correct handler

Figure 8–1 shows the algorithm that HAL top-level exception handler uses to distinguish between hardware interrupts and software exceptions.

Figure 8–1. HAL Top-Level Exception Handler


enter
Save context
Hardware interrupts enabled?
Yes
No
Hardware interrupts pending?
Yes
No
Handle software exception
Handle hardware interrupts
ISR 0
ISR 1
• • •
ISR 31
Restore context
exit

The top-level exception handler looks at the `estatus` register to determine the interrupt enable status. If the `EPIE` bit is set, hardware interrupts were enabled at the time the exception happened. If so, the exception handler looks at the IRQ bits in `ipending`. If any IRQs are asserted, the exception handler calls the hardware interrupt handler.

If hardware interrupts are not enabled at the time of the exception, it is not necessary to look at `ipending`.

If no IRQs are active, there is no hardware interrupt, and the exception is a software exception. In this case, the top-level exception handler calls the software exception handler.

All hardware interrupts are higher priority than software exceptions.

For details on the Nios II processor `estatus` and `ipending` registers, see the *Programming Model* chapter of the *Nios II Processor Reference Handbook.*

Upon return from the hardware interrupt or software exception handler, the top-level exception handler does the following:

1. Restores the stack pointer, if a private exception stack is used
2. Restores the registers from the stack
3. Exits by issuing an `eret` (exception return) instruction

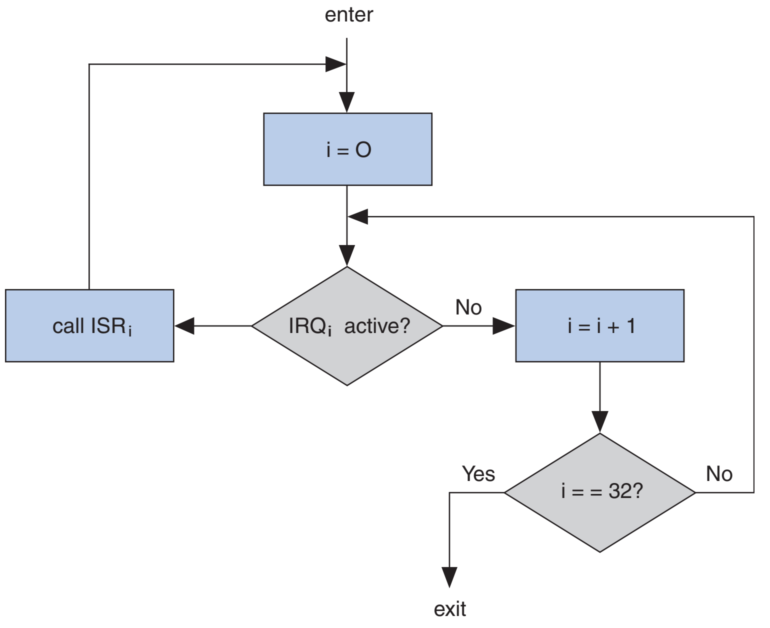
Hardware Interrupt Handler

The Nios II processor supports thirty-two hardware interrupts. In the HAL exception handler, hardware interrupt 0 has the highest priority, and 31 the lowest. This prioritization is a feature of the HAL exception handler, and is not inherent in the Nios II exception and interrupt controller.

The hardware interrupt handler calls the user-registered ISRs. It goes through the IRQs in `ipending` starting at 0, and finds the first (highest priority) active IRQ. Then it calls the corresponding registered ISR. After this ISR executes, the exception handler begins scanning the IRQs again, starting at IRQ_0. In this way, higher priority exceptions are always processed before lower-priority exceptions. When all IRQs are clear, the hardware interrupt handler returns to the top level. Figure 8–2 shows a flow diagram of the HAL hardware interrupt handler.

When the interrupt vector custom instruction is present in the Nios II processor, the HAL source detects it at compile time and generates code using the custom instruction. For further information, see "Use the Interrupt Vector Custom Instruction" on page 8–14.

Figure 8–2. HAL Hardware Interrupt Handler



Software Exception Handler

Software exceptions can include unimplemented instructions, traps, and other exceptions.

Software exception handling depends on options selected in the Nios II IDE. If you have enabled unimplemented instruction emulation, the exception handler first checks to see if an unimplemented instruction caused the exception. If so, it emulates the instruction. Otherwise, it handles traps and other exceptions.

Unimplemented Instructions

You can include a handler to emulate unimplemented instructions. The Nios II processor architecture defines the following implementation-dependent instructions:

- `mul`
- `muli`
- `mulxss`
- `mulxsu`

- `mulxuu`
- `div`
- `divu`

For details on unimplemented instructions, see the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook.*

Unimplemented instructions are different from invalid instructions, which are described in "Invalid Instructions" on page 8–23.

When to Use the Unimplemented Instruction Handler

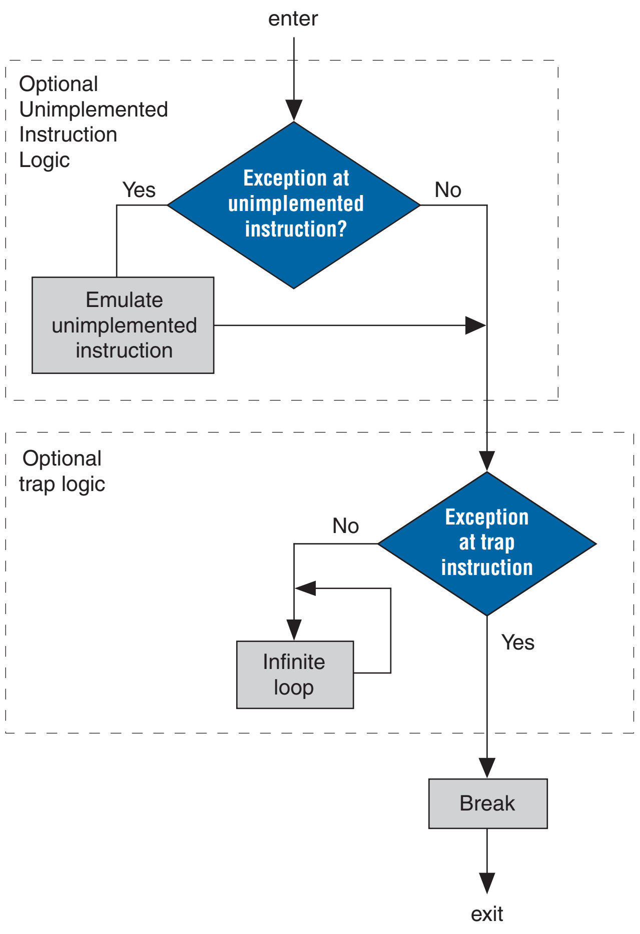
You do not normally need the unimplemented instruction handler, because the Nios II IDE includes software emulation for unimplemented instructions from its run-time libraries if you are compiling for a Nios II processor that does not support the instructions.

Here are the circumstances under which you might need the unimplemented instruction handler:

- You are running a Nios II program on an implementation of the Nios II processor other than the one you compiled for. The best solution is to build your program for the correct Nios II processor implementation. Only if this is not possible should you resort to the unimplemented instruction handler.
- You have assembly language code that uses an implementation-dependent instruction.

Figure 8–3 shows a flowchart of the HAL software exception handler, including the optional instruction emulation logic. If instruction emulation is not enabled, this logic is omitted.

Figure 8–3. HAL Software Exception Handler


enter
Optional Unimplemented Instruction Logic
Exception at unimplemented instruction?
Yes
No
Emulate unimplemented instruction
Optional trap logic
Exception at trap instruction
No
Yes
Infinite loop
Break
exit

If unimplemented instruction emulation is disabled, but the processor encounters an unimplemented instruction, the exception handler treats resulting exception as an other exception. Other exceptions are described in "Other Exceptions" on page 8–22.

Using the Unimplemented Instruction Handler

The unimplemented instruction handler defines an emulation routine for each of the implementation-dependent instructions. In this way, the full Nios II instruction set is always supported, even if a particular Nios II core does not implement all instructions in hardware.

To include the unimplemented instruction handler, turn on **Emulate multiply and divide instructions** on the **System properties** page of the Nios II IDE. The emulation routines are small (less than ¾ KBytes of memory), so it is usually safe to include them even when targeting a Nios II core that does not require them. If a Nios II core implements a particular instruction in hardware, its corresponding exception never occurs.

☞ An exception routine must never execute an unimplemented instruction. The HAL exception handling system does not support nested software exceptions.

Software Trap Handling

If the cause of the software exception is not an unimplemented instruction, the HAL software exception handler checks for a `trap` instruction. The HAL is not designed to handle software traps. If it finds one, it executes a `break`.

If your software is compiled for release, the exception handler makes a distinction between traps and other exceptions. If your software is compiled for debug, traps and other exceptions are handled identically, by executing a `break` instruction. Figure 8–3 shows a flowchart of the HAL software exception handler, including the optional trap logic. If your software is compiled for debug, the trap logic is omitted.

In the Nios II IDE, you can select debug or release compilation in the **Project Properties** dialog box, under **C/C++ Build**.

Other Exceptions

If the exception is not caused by an unimplemented instruction or a trap, it is an other exception. In a debugging environment, the processor executes a `break`, allowing the debugger to take control. In a non-debugging environment, the processor goes into an infinite loop.

For details about the Nios II processor `break` instruction, see the *Programming Model* and *Instruction Set Reference* chapters of the *Nios II Processor Reference Handbook*.

Other exceptions can occur for these reasons:

- You need to include the unimplemented instruction handler, discussed in "Unimplemented Instructions" on page 8–19.

- A peripheral is generating spurious interrupts. This is a symptom of a serious hardware problem. A peripheral might generate spurious hardware interrupts if it deasserts its interrupt output before an ISR has explicitly serviced it.

Invalid Instructions

An invalid instruction word contains invalid codes in the OP or OPX field. For normal Nios II core implementations, the result of executing an invalid instruction is undefined; processor behavior is dependent on the Nios II core.

Therefore, the exception handler cannot detect or respond to an invalid instruction.

Invalid instructions are different from unimplemented instructions, which are described in "Unimplemented Instructions" on page 8–19.

For more information, see the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

HAL Exception Handler Files

The HAL exception handling code is in the following files:

- Source files:
  - alt_exception_entry.S
  - alt_exception_muldiv.S
  - alt_exception_trap.S
  - alt_irq_entry.S
  - alt_irq_handler.c
  - alt_software_exception.S
  - alt_irq_vars.c
  - alt_irq_register.c
- Header files:
  - alt_irq.h
  - alt_irq_entry.h

Referenced Documents

This chapter references the following documents:

- *Programming Model* chapter of the *Nios II Processor Reference Handbook*
- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*
- *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*

- *HAL API Reference* chapter in the *Nios II Software Developer's Handbook*
- *Quartus II Handbook, Volume 5: Embedded Peripherals*
- *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*
- *Using Nios II Tightly Coupled Memory Tutorial*
- *DMA Controller Core* chapter of the *Quartus II Handbook, Volume 5: Embedded Peripherals*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*
- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*

Document Revision History

Table 8–2 shows the revision history for this document.

Table 8–2. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	• Chapter 7 was formerly chapter 6. • Added table of contents to Introduction section. • Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	• Describes support for the interrupt vector custom instruction.	Interrupt vector custom instruction added.
May 2006 v6.0.0	• Corrected error in `alt_irq_enable_all()` usage • Added illustrations • Revised text on optimizing ISRs • Expanded and revised text discussing HAL exception handler code structure.	
October 2005 v5.1.0	• Updated references to HAL exception-handler assembly source files in section "HAL Exception Handler Files". • Added description of `alt_irq_disable()` and `alt_irq_enable()` in section "ISRs".	
May 2005 v5.0.0	Added tightly-coupled memory information.	
December 2004 v1.2	Corrected the "Registering the Button PIO ISR with the HAL" example.	
September 2004 v1.1	• Changed examples. • Added ISR performance data.	
May 2004 v1.0	Initial Release.	