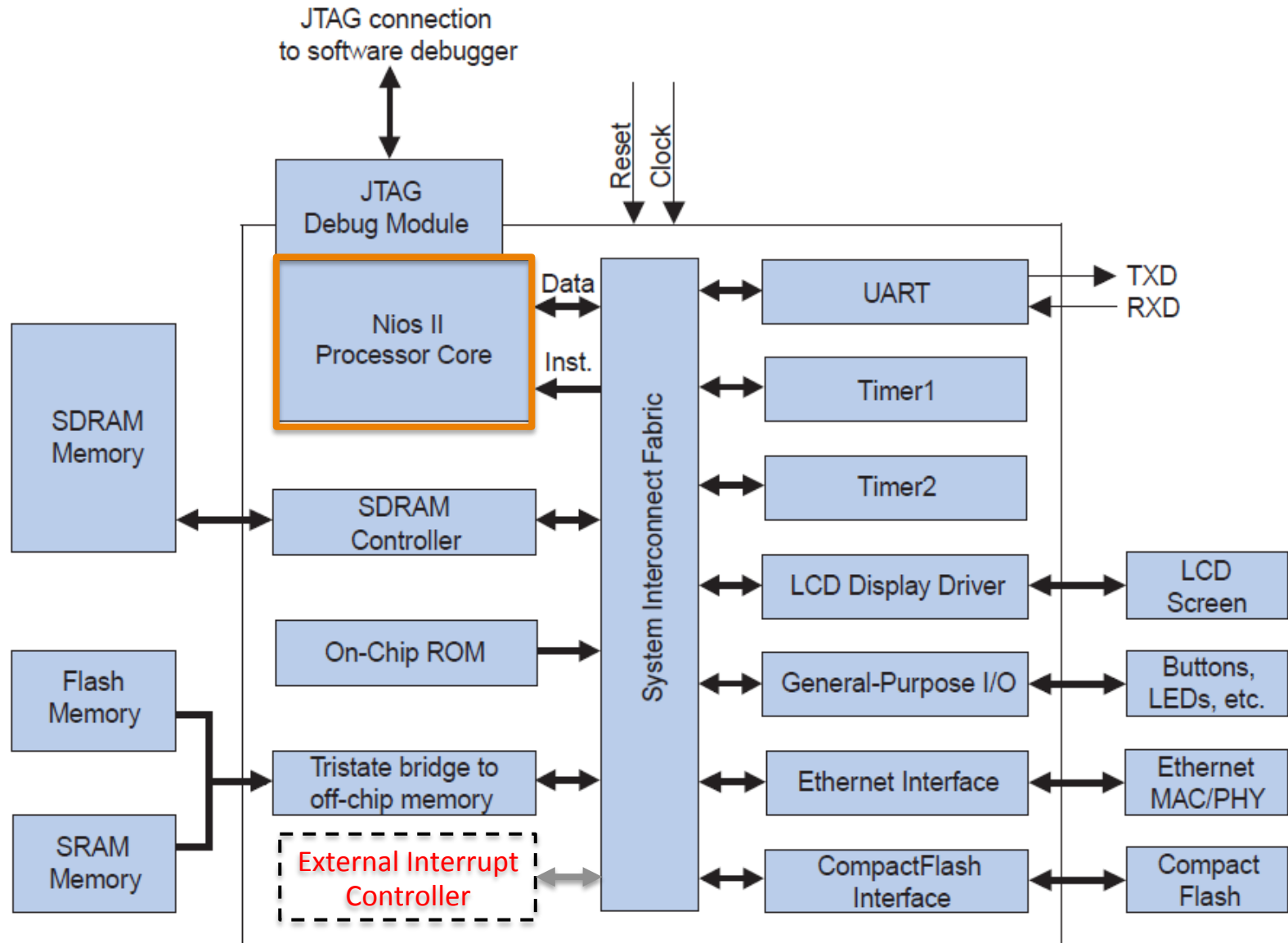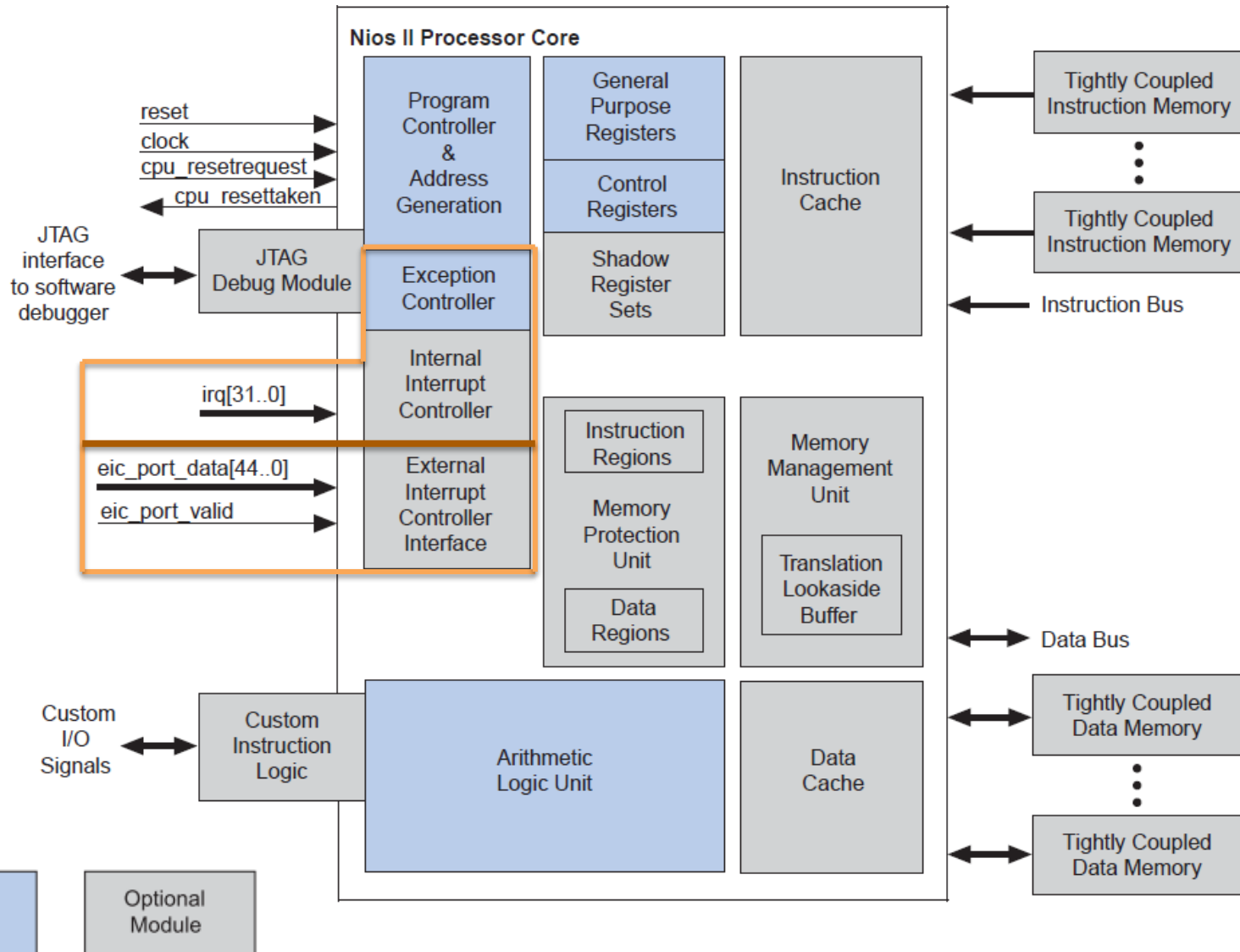# SISTEMI EMBEDDED
AA 2014/2015

(Software) Exceptions and (Hardware) Interrupts

Federico Baronti

Example of a Nios II System

Nios II Processor Core Architecture

Reset signals

reset	Reset	This is a global hardware reset signal that forces the processor core to reset immediately.
cpu_resetrequest	Reset	This is an optional, local reset signal that causes the processor to reset without affecting other components in the Nios II system. The processor finishes executing any instructions in the pipeline, and then enters the reset state. This process can take several clock cycles, so be sure to continue asserting the cpu_resetrequest signal until the processor core asserts a cpu_resettaken signal.

The processor core asserts a cpu_resettaken signal for 1 cycle when the reset is complete and then periodically if cpu_resetrequest remains asserted. The processor remains in the reset state for as long as cpu_resetrequest is asserted. While the processor is in the reset state, it periodically reads from the reset address. It discards the result of the read, and remains in the reset state.

The processor does not respond to cpu_resetrequest when the processor is under the control of the JTAG debug module, that is, when the processor is paused. The processor responds to the cpu_resetrequest signal if the signal is asserted when the JTAG debug module relinquishes control, both momentarily during each single step as well as when you resume execution. |

Exceptions and Interrupts

- **Exception**: a transfer of control away from a program's normal flow of execution, caused by an event, <u>either internal or external to the processor</u>, **which requires immediate attention**

- **Interrupt**: an exception caused by an explicit request signal from an external device (hardware/interrupt exception)

Exception types (1)

- **Reset exception**: occurs when the Nios II processor is reset. Control is transferred to the _reset address_ specified when generating the Nios II processor core

- **Break exception**: occurs when the JTAG debug module requests control. Control is transferred to the _break address_ specified when generating the Nios II processor core

Exception types (2)

- **Instruction-related exception:** occurs when one of several internal conditions occurs. Control is transferred to the *general exception address* specified when generating the Nios II processor core (Software exception)

- **Interrupt exception**: occurs when a peripheral device signals a condition requiring service. Control is transferred to the *general exception address*

Break exceptions

- A break is a transfer of control away from a program's normal flow of execution for the purpose of debugging

- Software debugging tools can take control of the Nios II processor via the JTAG debug module to implement debug and diagnostic features, such as breakpoints and watchpoints

- The processor enters the break processing state under one of the following conditions:
  - The processor executes the break instruction (software break)
  - The JTAG debug module asserts a hardware break

Instruction-related exceptions

- Occur during execution of Nios II instructions
  - Trap instruction: software-invoked exception. Useful to "call" OS services without knowing the routine run-time addresses
  - Break Instruction
  - Illegal instruction
  - Unimplemented instruction
  - Division error
  - …

Interrupt exceptions

- A peripheral device can request an interrupt by asserting an interrupt request (IRQ) signal. IRQs interact with the Nios II processor through an <u>interrupt controller</u>

- The Nios II processor can be configured with one of the following <u>interrupt controller options</u>:

  – The internal interrupt controller

  – The external interrupt controller interface

Some definitions

- **Exception (interrupt) latency:** the time elapsed between the event that causes the exception (assertion of an interrupt request) and the execution of the first instruction at the handler address

- **Exception (interrupt) response time**: the time elapsed between the event that causes the exception (assertion of an interrupt request) and the execution of non-overhead exception code, which is specific to the exception type (device)

Internal interrupt controller

- **Non-vectored** exception controller to handle all exception types

- Each exception, including hardware interrupts (IRQ31-0), causes the processor to transfer execution to the same *general exception address*

- An exception handler at this address determines the cause of the exception and dispatches an appropriate exception routine

External interrupt controller interface (1)

- **External Interrupt Controller (EIC) can be used to shorten exception response time**
- EIC can monitor and prioritize IRQ signals and determine which interrupt to present to the Nios II processor. An EIC can be software-configurable
- When an IRQ is asserted, the EIC provides the following data to the Nios II processor:
  - **The requested handler address (RHA)**
  - The Requested Interrupt Level (RIL); the interrupt is taken only when the RIL is greater than the IL field (6-bit) in the status register
  - The Requested Register Set (RRS)
 - Requested NonMaskable Interrupt (RNMI) mode

External interrupt controller interface (2)

- Requested register set is one of the implemented shadow register sets
  - This way the context switch overhead is eliminated (useful for high-critical interrupts)
  - Less critical interrupts can share the same shadow register set
    - No problem if interrupt pre-emption cannot occur among these interrupts
      - Same priority level or nested interrupts are disabled
    - Otherwise the ISR must save its register set on entry and restore it on exit

External interrupt controller interface (3)

- The Nios II processor EIC interface connects to a single EIC, but an EIC can support a daisy-chained configuration

- Multiple EICs can monitor and prioritize interrupts

- The EIC directly connected to the processor presents the processor with the highest-priority interrupt from all EICs in the daisy chain

- An EIC component can support an arbitrary level of daisy-chaining, potentially allowing the Nios II processor to handle an arbitrary number of prioritized interrupts

External interrupt controller interface (4)
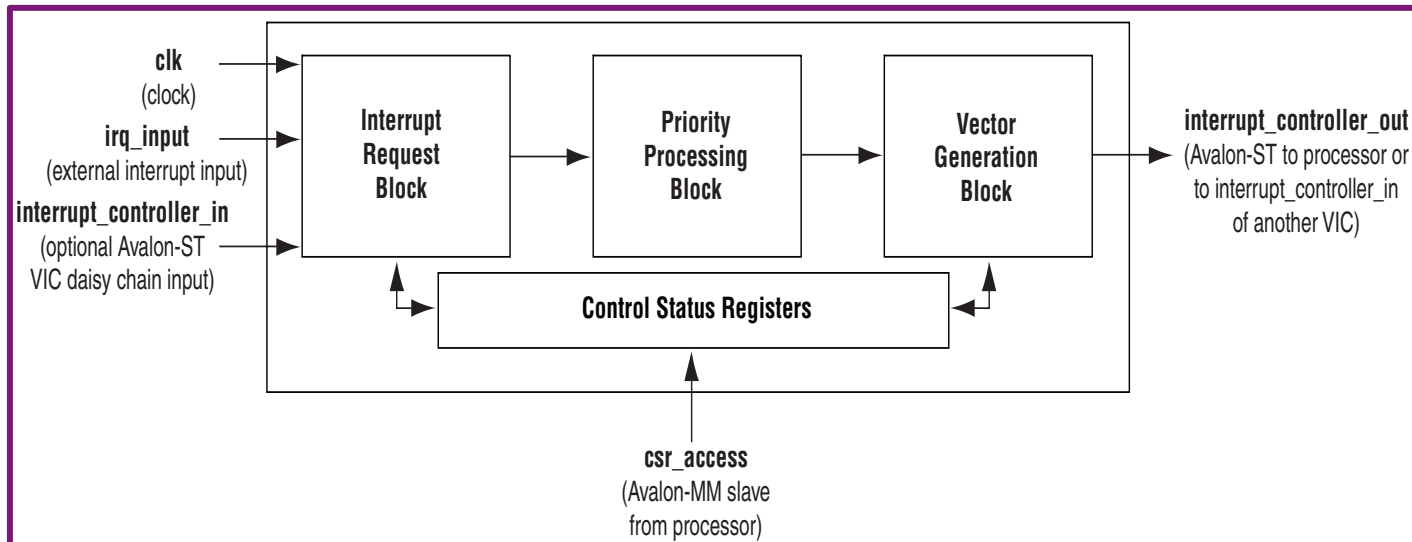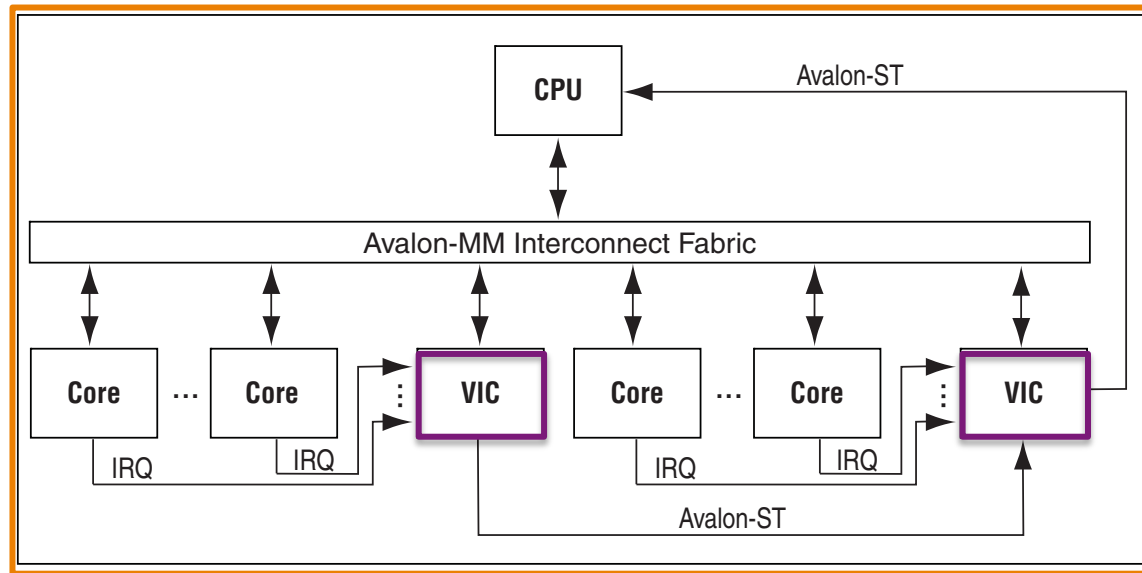
Nios II registers (1)

- **General-purpose registers (r0-r31)**

•••

Register	Name	Function
r20		Callee-saved register
r21		Callee-saved register
r22		Callee-saved register
r23		Callee-saved register
r24	et	Exception temporary
r25	bt	Breakpoint temporary (1)
r26	gp	Global pointer
r27	sp	Stack pointer
r28	fp	Frame pointer
r29	ea	Exception return address
r30	ba	Breakpoint return address (2)
r31	ra	Return address

Nios II registers (2)

- **Control registers** accessible only by the special instructions *rdctl* and *wrctl* that are only available in supervisor mode

Register	Name	Register Contents
0	status	Refer to Table 3–7 on page 3–12
1	estatus	Refer to Table 3–9 on page 3–14
2	bstatus	Refer to Table 3–10 on page 3–15
3	ienable	Internal interrupt-enable bits *(3)*
4	ipending	Pending internal interrupt bits *(3)*
5	cpuid	Unique processor identifier
6	Reserved	Reserved
7	exception	Refer to Table 3–12 on page 3–16
8	pteaddr *(1)*	Refer to Table 3–13 on page 3–16
9	tlbacc *(1)*	Refer to Table 3–15 on page 3–17
10	tlbmisc *(1)*	Refer to Table 3–17 on page 3–18
11	Reserved	Reserved
12	badaddr	Refer to Table 3–19 on page 3–21
13	config *(2)*	Refer to Table 3–21 on page 3–21
14	mpubase *(2)*	Refer to Table 3–23 on page 3–22
15	mpuacc *(2)*	Refer to Table 3–25 on page 3–23
16–31	Reserved	Reserved

Status register (1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								RSIE	NMI	PRS						CRS					IL								IH	EH	U	PIE

Bit	Description	Access	Reset	Available
RSIE	RSIE is the register set interrupt-enable bit. When set to 1, this bit allows the processor to service external interrupts requesting the register set that is currently in use. When set to 0, this bit disallows servicing of such interrupts.	Read/Write	1	EIC interface and shadow register sets only *(4)*
NMI	NMI is the nonmaskable interrupt mode bit. The processor sets NMI to 1 when it takes a nonmaskable interrupt.	Read	0	EIC interface only *(3)*
PRS	PRS is the previous register set field. The processor copies the CRS field to the PRS field upon one of the following events: ■ In a processor with no MMU, on any exception ■ In a processor with an MMU, on one of the following: ■ Break exception ■ Nonbreak exception when status.EH is zero The processor copies CRS to PRS immediately after copying the status register to estatus, bstatus or sstatus. The number of significant bits in the CRS and PRS fields depends on the number of shadow register sets implemented in the Nios II core. The value of CRS and PRS can range from 0 to *n*-1, where *n* is the number of implemented register sets. The processor core implements the number of significant bits needed to represent *n*-1. Unused high-order bits are always read as 0, and must be written as 0. ☞ Ensure that system software writes only valid register set numbers to the PRS field. Processor behavior is undefined with an unimplemented register set number.	Read/Write	0	Shadow register sets only *(3)*

Status register (2)

| Bit	Description	Access	Reset	Available
CRS	CRS is the current register set field. CRS indicates which register set is currently in use. Register set 0 is the normal register set, while register sets 1 and higher are shadow register sets. The processor sets CRS to zero on any noninterrupt exception. The number of significant bits in the CRS and PRS fields depends on the number of shadow register sets implemented in the Nios II core. Unused high-order bits are always read as 0, and must be written as 0.	Read *(1)*	0	Shadow register sets only *(3)*
IL	IL is the interrupt level field. The IL field controls what level of external maskable interrupts can be serviced. The processor services a maskable interrupt only if its requested interrupt level is greater than IL.	Read/Write	0	EIC interface only *(3)*
IH	IH is the interrupt handler mode bit. The processor sets IH to one when it takes an external interrupt.	Read/Write	0	EIC interface only *(3)*
EH *(2)*	EH is the exception handler mode bit. The processor sets EH to one when an exception occurs (including breaks). Software clears EH to zero when ready to handle exceptions again. EH is used by the MMU to determine whether a TLB miss exception is a fast TLB miss or a double TLB miss. In systems without an MMU, EH is always zero.	Read/Write	0	MMU only *(3)*
U *(2)*	U is the user mode bit. When U = 1, the processor operates in user mode. When U = 0, the processor operates in supervisor mode. In systems without an MMU, U is always zero.	Read/Write	0	MMU or MPU only *(3)*
PIE	PIE is the processor interrupt-enable bit. When PIE = 0, internal and maskable external interrupts and noninterrupt exceptions are ignored. When PIE = 1, internal and maskable external interrupts can be taken, depending on the status of the interrupt controller. Noninterrupt exceptions are unaffected by PIE.	Read/Write	0	Always

Other control registers (1)

- The **estatus register** holds a saved copy of the status register during nonbreak exception processing

- The **bstatus register** holds a saved copy of the status register during break exception processing

- The **ienable register** controls the handling of internal hardware interrupts

- The **ipending register** indicates the value of the interrupt signals driven into the processor

Other control registers (2)

- The **cpuid register** holds a constant value that is defined in the Nios II Processor parameter editor to uniquely identify each processor in a multiprocessor system

- When the extra exception information option is enabled, the Nios II processor provides information useful to system software for exception processing in the *exception* and *badaddr* registers when an exception occurs

- …

Masking and disabling interrupts

with Internal Interrupt Controller
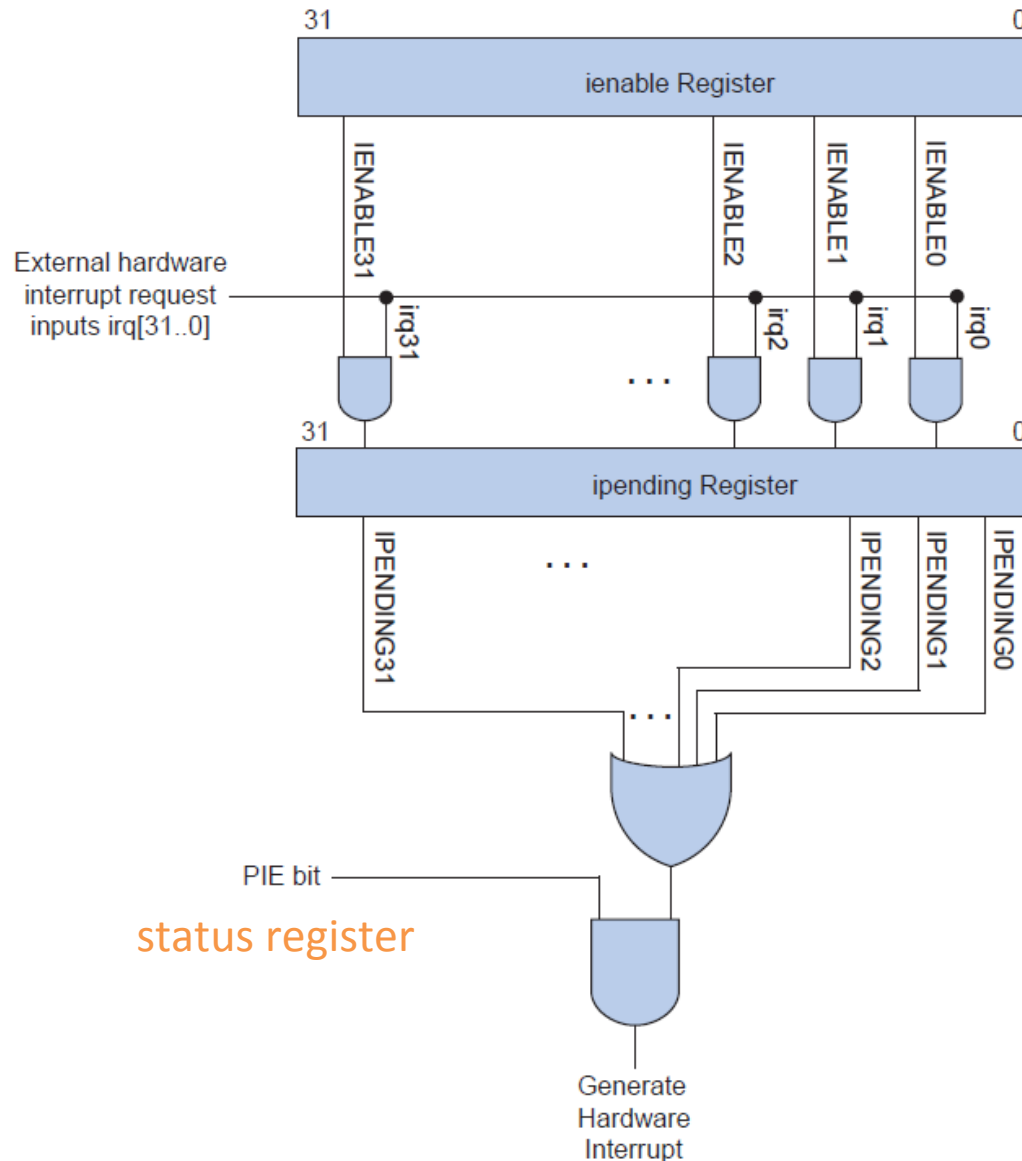
Exception processing flow

- The <u>general exception handler</u> is a routine **that determines the cause of each exception** and then dispatches an exception routine to respond to the specific exception (software or hardware)
- The <u>general exception handler</u> is found at the *general exception address*
  - At run time this address is fixed, and software cannot modify it
  - Programmers do not directly access exception vectors and can write programs without awareness of this address

Determining the exception cause

- Instruction-related exception
  - *cause* filed of the *exception* register (if present) stores the info on what instruction has caused the exception
  - If non-present, the handler must retrieve the instruction that has caused the exception

Pseudo C code for dispatiching software exceptions (w/o *excepetion* register) and hardware interrupts

```
/* With an internal interrupt controller, check for interrupt exceptions. With an external interrupt
 * controller, ipending is always 0, and this check can be omitted. */
if (estatus.PIE == 1 and ipending != 0)  handle hardware interrupt
else {
        /* Decode exception from instruction */
        decode instruction at [ea]-4
        if (instruction is trap) handle trap exception
        else if (instruction is load or store) handle misaligned data address exception
        else if (instruction is branch, bret, callr, eret, jmp, or ret)
                handle misaligned destination address exception
        else if (instruction is unimplemented) handle unimplemented instruction exception
        else if (instruction is illegal) handle illegal instruction exception
        else if (instruction is divide) {
                if (denominator == 0) handle division error exception
                else if (instruction is signed divide and numerator == 0x80000000
                                                and denominator == 0xffffffff)
                        handle division error exception
                }
        /* Not any known exception */
        else  handle unknown exception
        }
```

Hardware interrupts processing flow w/ EIC

- When the EIC interface presents an interrupt to the Nios II processor, the processor uses several criteria, as follows, to determine whether to take the interrupt:
  - **Nonmaskable interrupts:** the processor takes any NMI as long as it is not processing a previous NMI
  - **Maskable interrupts**: the processor takes a maskable interrupt if maskable interrupts are enabled, and if the requested interrupt level is higher than that of the interrupt currently being processed (if any)
    - However, if shadow register sets are implemented, the processor takes the interrupt only if the interrupt requests a register set different from the current register set, or if the register set interrupt enable flag (status.RSIE) is set

Nested exceptions (1)

- Nested exceptions can occur under the following circumstances:
  - An exception handler enables maskable interrupts
  - An EIC is present and an NMI occurs
  - An EIC is present and the processor is configured to keep maskable interrupts enabled when taking an interrupt
  - An exception handler triggers an instruction-related exception

Nested exceptions (2)

- By default, Nios II processor disables maskable interrupts when it takes an interrupt request

- To enable nested interrupts, the ISR itself must re-enable interrupts after the interrupt is taken

- Alternatively, to take full advantage of nested interrupts with shadow register sets, system software <u>can set the config.ANI flag</u>. When config.ANI = 1, the Nios II processor keeps maskable interrupts enabled after it takes an interrupt

Interrupt Service Routine (ISR)

- The HAL provides an **enhanced application programming interface** (API) for writing, registering and managing ISRs
  - This API is compatible with both internal and external hardware interrupt controllers
- For back compatibility Altera also supports a **legacy** hardware interrupt API
  - This API supports only the IIC
  - A custom driver written prior to Nios II version 9.1 uses the legacy API

HAL API

- Both interrupt APIs include the following types of routines:
  - Routines to be called by a device driver to register an ISR
  - Routines to be called by an ISR to manage its environment
  - Routines to be called by BSP or application code to control ISR behavior
- Both interrupt APIs support the following types of BSPs:
  - HAL BSP without an RTOS
  - HAL-based RTOS BSP, such as a MicroC/OS-II BSP
- When an EIC is present, the controller's driver provides functions to be called by the HAL

HAL API selection

- When the SBT creates a BSP, it determines whether the BSP must implement the legacy interrupt API
  - Each driver that supports the enhanced API publishes this capability to the SBT through its <driver name>_sw.tcl file
- **The BSP implements the enhanced API if all drivers support it; otherwise it uses the legacy API**
  - Altera drivers written for the enhanced API, also support the legacy one
  - Devices whose interrupts are not connected to the Nios II processor are ignored

Example DE2 Basic Computer

- **system.h**

```
/*
 * System configuration
 */
#define ALT_DEVICE_FAMILY "CYCLONEII"
#define ALT_IRQ_BASE NULL
#define ALT_LEGACY_INTERRUPT_API_PRESENT
#define ALT_LOG_PORT "/dev/null"
#define ALT_LOG_PORT_BASE 0x0
#define ALT_LOG_PORT_DEV null
#define ALT_LOG_PORT_TYPE ""
#define ALT_NUM_EXTERNAL_INTERRUPT_CONTROLLERS 0
#define ALT_NUM_INTERNAL_INTERRUPT_CONTROLLERS 1
#define ALT_NUM_INTERRUPT_CONTROLLERS 1
```

avalon_parallel_port_driver and up_avalon_rs232_driver do not support enhanced API

Enhanced HAL Interrupt API

Function Name	Implemented By
`alt_ic_isr_register()`	Interrupt controller driver *(1)*
`alt_ic_irq_enable()`	Interrupt controller driver *(1)*
`alt_ic_irq_disable()`	Interrupt controller driver *(1)*
`alt_ic_irq_enabled()`	Interrupt controller driver *(1)*
`alt_irq_disable_all()`	HAL
`alt_irq_enable_all()`	HAL
`alt_irq_enabled()`	HAL

Note to Table 8–1:

(1) If the system is based on an EIC, these functions must be implemented by the EIC driver. If the system is based in the IIC, the functions are implemented by the HAL. For details about each function, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

- Using the enhanced HAL API to implement ISRs requires performing the following steps:
  - Write the ISR that handles hardware interrupts for a specific device
  - Ensure that the main program registers the ISR with the HAL by calling the alt_ic_isr_register() function (this function also enables the hardware interrupts)

Legacy HAL Interrupt API

- 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()

- Using the legacy HAL API to implement ISRs requires performing the following steps:
  - Write the ISR that handles hardware interrupts for a specific device
  - Ensure that the main program registers the ISR with the HAL by calling the alt_irq_register() function
  - alt_irq_register() enables also hardware interrupts by calling alt_irq_enable_all()

HAL exception handling w/ IIC
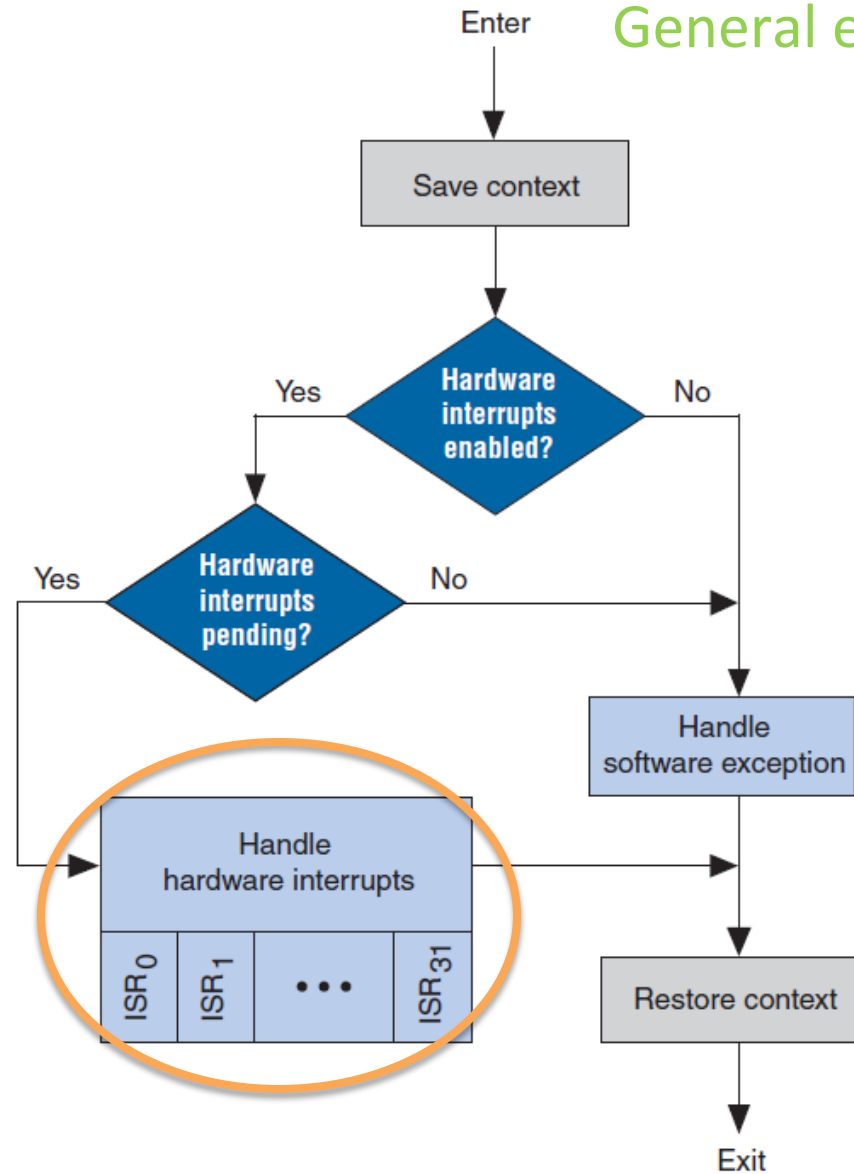
Harwdare interrupt funnel

After the ISR$_i$ execution, ipending register is scanned again from 0, so that higher-priority interrupts are always processed before lower-priority interrupts

In the HAL funnel, hardware interrupt 0 has the highest priority, and 31 the lowest priority

ISR code must clear the associated peripheral's interrupt condition

Enter

i = O

Call ISR$_i$ ← Yes — IRQ$_i$ active? — No → i = i + 1

Yes — i == 32? — No

Exit
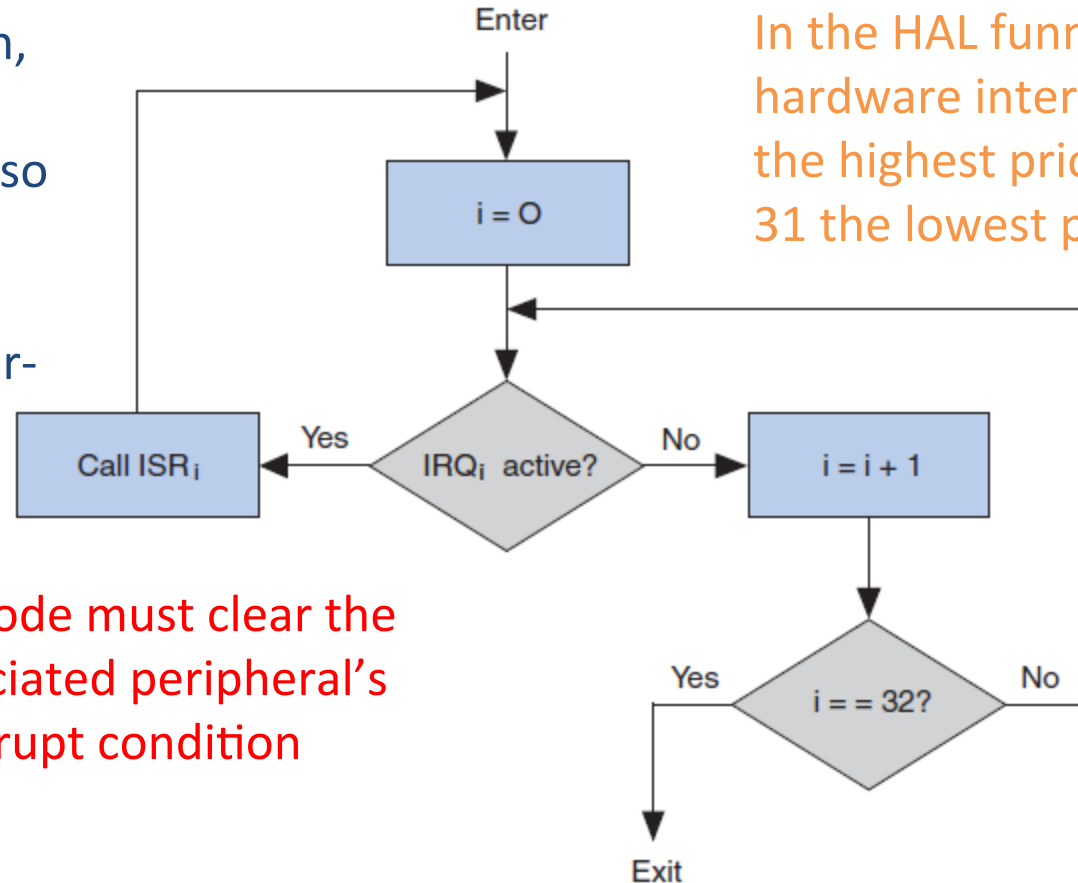
When writing an ISR…

- 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, never include function calls that can block for any reason (such as waiting for a hardware interrupt)
  - Avoid using the C standard library I/O API, because calling these functions can result in deadlock within the system, that is, the system can become permanently blocked in the ISR
  - 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 a hardware interrupt that never occurs because interrupts are disabled

Putting into practice (1)

- Write a program that reads the pushbutton activity **exploting the related hardware interrupt** and turns on/off some LEDs

- #include <sys/alt_irq.h> to use Interrupt HAL API

- ISR prototype
  - static void pushbutton_ISR(void* context, unsigned long id);

Putting into practice (2)

- Make GREEN leds blink using the Interval Timer and the **sys_clk HAL** w/ 2 s period
  - Map **sys_clk HAL** to the Interval_timer peripheral using the BSP editor
  - Define a variable of *alt_alarm* type (you need to include **"sys/alt_alarm.h "** header file)
  - Start the alarm using the *alt_alarm_start*() function passing as parameter the pointer to the callback function that makes the leds blink
    - Prototype of the callback function:
      alt_u32 **my_alarm_callback(void* context)**
    - The return value is the time that will pass before the next alarm event
  - Handle the GREEN_LEDS Parallel Port using the related HAL; see "altera_up_avalon_parallel_port.h " header for how to use it

References

- Altera, "Nios II Processor Reference Handbook," *n2cpu_nii5v1.pdf*
  - 2. Processor Architecture
  - 3. Programming Model/Exception Processing
- Altera, "Nios II Software Developer's Handbook," *n2sw_nii5v2.pdf*
  - 8. Exception Handling
  - 14. HAL API Reference