Nios II Interruption Simulation

Learning Goal: Implement an interrupt handler.

Requirements: Nios2Sim Simulator.

1 Introduction

The goal of the first group of labs is to implement a stopwatch on FPGA which can accurately measure the time while reacting to the user's input. To do so, you will first develop the following components:

- a hardware timer (**lab 1**), which generates a periodic interrupt.
- interrupt service routines (ISRs) to handle the interrupts from the timer and from the buttons (**this lab**).
- a Nios II processor with interrupt support (**lab 3**).

which you will eventually use in **lab 4** to implement the stopwatch.

During this lab, you will write an interrupt handler with two possible sources. For this purpose, you will use the **Nios2Sim** Simulator. Make sure to download the last version of the simulator from the course Moodle. For more details about the Nios2Sim Simulator, please refer to the **Introduction to the Nios II Simulator** lab available under the **Additional material** section of the course Moodle. On the course Moodle, you can also find a summary of the instruction set of the Nios II processor (**Nios II Instruction Set Summary**).

2 The Nios II Control Registers

In addition to the 32 general purpose registers, the Nios II has six control registers. Most of these registers are used to control the interrupts. By using these registers you can activate or deactivate interrupts from a particular source, and find the interrupts that are currently pending.

You will need the two following instructions to read or write the control registers:

- `rdctl rC, ctl`*N* : reads the value of the control register *N* and stores it in the register `rC`.
- `wrctl ctl`*N*`, rA` : writes the value of the register `rA` in the control register *N*.

Note: In your assembly program, you can use the name of the control registers given in the following section instead of ctlN.

2.1 Description

This section describes the control registers related to interrupts.

Register	Name	31...1	0
ctl0	`status`	*Reserved*	PIE
ctl1	`estatus`	*Reserved*	EPIE
ctl2	`bstatus`	*Reserved*	BPIE
ctl3	`ienable`	Interrupt-enable bits	
ctl4	`ipending`	Pending-interrupt bits	
ctl5	`cpuid`	Unique processor identifier	

2.1.1 status

PIE is the processor interrupt-enable bit. When **PIE** is zero, interrupts are ignored. Write 1/0 to the **PIE** bit to enable/disable interrupts. The **PIE** bit is cleared at reset.

2.1.2 estatus

EPIE holds a saved copy of **PIE** during exception processing.

2.1.3 bstatus

The `bstatus` register is not used for interrupts.

2.1.4 ienable

The `ienable` register controls the handling of external hardware interrupts. Each bit of the `ienable` register corresponds to one of the interrupt inputs, irq*0* through irq*31*. A value of 1 in bit *n* means that the corresponding irq*n* interrupt is enabled; a value of 0 means that the corresponding interrupt is disabled.

2.1.5 ipending

The value of the `ipending` register indicates which interrupts are currently pending. A value of 1 in bit *n* means that the corresponding irq*n* input is asserted and that it is enabled (i.e., `ienable`*n* is 1). Writing a value to the `ipending` register has no effect (*read-only*).

2.1.6 cpuid

The `cpuid` register is not used for interrupts.

3 Writing an Interrupt Handler

When an interrupt occurs, the current instruction is aborted and replaced by a jump to the *interrupt handler* address. By default, the Nios II simulator assumes that the *interrupt handler* starts at the address `0x0004` (i.e., the second instruction line). Therefore, as shown in the following example, the first instruction line (i.e., the line being executed after a reset) must jump to the main function.

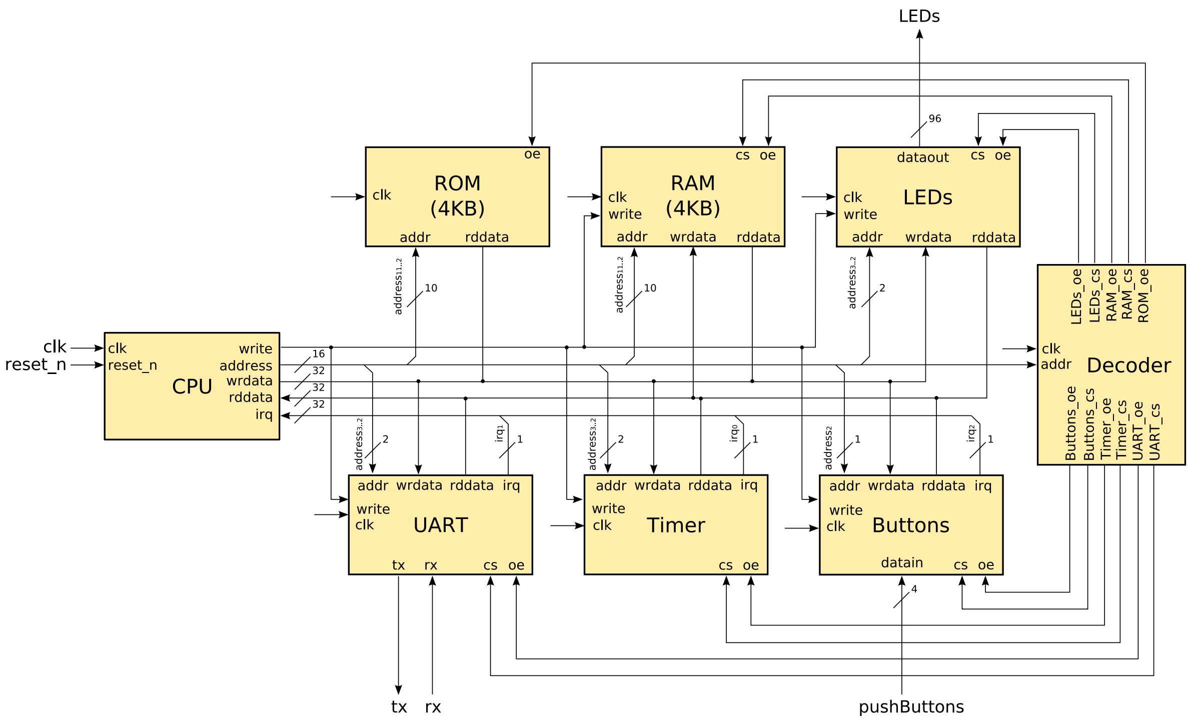
During an interrupt, the address of the next instruction is stored in the `ea` register (i.e., the *Exception Return Address* register). Before returning from the interrupt handler, the `ea` register must be decremented by 4 to execute the instruction that has been aborted previously.

Here is the general structure of a program that includes an interrupt handler.

```
_start:
    br    main        ; jump to the main function
interrupt_handler:
    ...               ; save the registers to the stack
    ...                 ; read the ipending register to identify the source
    ...                 ; call the corresponding routine
    ...                 ; restore the registers from the stack
    addi  ea, ea, -4    ; correct the exception return address
    eret                ; return from exception
main:
    ...                 ; main procedure here
```

4 Description of the System

This section describes the peripherals that are connected to the CPU for this exercise. The following figure gives an overview of the system.



4.1 Memory Map

0x0000	0x0FFF	ROM
0x1000	0x1FFF	RAM
0x2000	0x200F	LEDs
0x2010	0x201F	UART
0x2020	0x202F	Timer
0x2030	0x2037	Buttons
0x2038	0xFFFF	

IRQ bit	IRQ Source
0	Timer
1	UART
2	Buttons

4.1.1 LEDs (0x2000 - 0x200F)

The **LEDs** module is an output interface that gives you access to the LEDs of the FPGA4U. This interface has three internal registers to specify the pattern to display on the LEDs and a register to modify the intensity of the LEDs.

Register	Address	Name	31 . . . 8	7 . . . 0
0	`0x2000`	`leds0`	LEDs (31 . . . 0)	
1	`0x2004`	`leds1`	LEDs (63 . . . 32)	
2	`0x2008`	`leds2`	LEDs (95 . . . 64)	
3	`0x200C`	`pulsewidth`	*Reserved*	Pulse Width

- The `leds0-2` registers are connected to the LEDs.
- The 8-bit `pulsewidth` is used to choose the intensity of the LEDs. The value of the register specifies the number of cycles during which the pattern on the LEDs is enabled over a 256-cycle period (i.e., the larger, the brighter).

4.1.2 UART (0x2010 - 0x201F, IRQ$_1$)

You will not use this peripheral for this experiment.

4.1.3 Timer (0x2020 - 0x202F, IRQ$_0$)

This is the timer that you designed in the lab *Timer: A Peripheral Interface*.

4.1.4 Buttons (0x2030 - 0x2037, IRQ$_2$)

The **Buttons** module is an input interface connected to the four push buttons of the FPGA4U. Each time one of the buttons is pressed, the module generates an interrupt.

The interface has two registers:

Register	Address	Name	31...4	3 . . . 0
0	`0x2030`	`status`	*Reserved*	State of the Buttons
1	`0x2034`	`edgecapture`	*Reserved*	Falling edge detection

- The `status` register shows the current state of the push buttons. If the `status`*N* bit is set to 0/1, it means that the push button *N* is pressed/released.
- When the push button *N* is pressed, a falling edge is detected and the `edgecapture`*N* bit is set to 1. The bit stays set until it is explicitly cleared by writing 0 to it. If the `edgecapture` register is non-zero, the **Buttons** module generates an IRQ. Write in the `edgecapture` register to clear it.

5 Exercise

- Download the last version of the **Nios2Sim** simulator.
  - Since version 0.4, the simulator supports the `.equ symbol, expression` directive and is able to resolve arithmetic expressions that includes symbols. The `.equ` directive sets the value of *symbol* to *expression*. Here is an example of what you can write:

```
.equ  TIMER, 0x2020     ; timer address
...
ldw  t0, TIMER+12(zero) ; read counter
```

- Note that **Nios2Sim** supports many instructions, however **only** instructions found in the "Nios 2 Instruction-Set Summary" (available on the course website) should be used.
- Implement a program that displays three counters on the LEDs:
  - The first counter is controlled from the main procedure in an infinite loop.
  - The second counter must be incremented each time the timer generates an IRQ. Set the period such that the counter is incremented once every 100 cycles.
  - The third counter must be changed each time that one of the push buttons is pressed. The first button (button 0) should increment the counter and the second button should decrement the counter. You are free to give different functionalities for the other two push buttons (e.g., increment, decrement, larger step, etc.).
  - Implement an interrupt handler that looks for the source of the interrupt and calls the corresponding interrupt service routine (*ISR*). Do not forget to consider what should happen if there are two interrupts in the same cycle.
  - Implement two *ISRs* to handle the interrupts coming from the timer and the push buttons.
  - In the main procedure, initialize the stack pointer and set the value of the control registers to enable the interrupts.
- Simulate your program. (Simulation >Start Simulation)
  - During simulation, you can set breakpoints by double-clicking on a line.
  - In the **Buttons** tab, you can use the toggle buttons to modify the `status` register.
  - You can observe the value of the control registers and the interrupt source vector in the **Control** tab.

6 Submission

In the lab 4 submission, submit the file created in the Exercise section with the following file name: `ihandler.asm`.