Nios II: Interrupt Management

Learning Goal: Implement a way to manage interrupts in hardware.

Requirements: ModelSim, Quartus, Nios2Sim Simulator.

1 Introduction

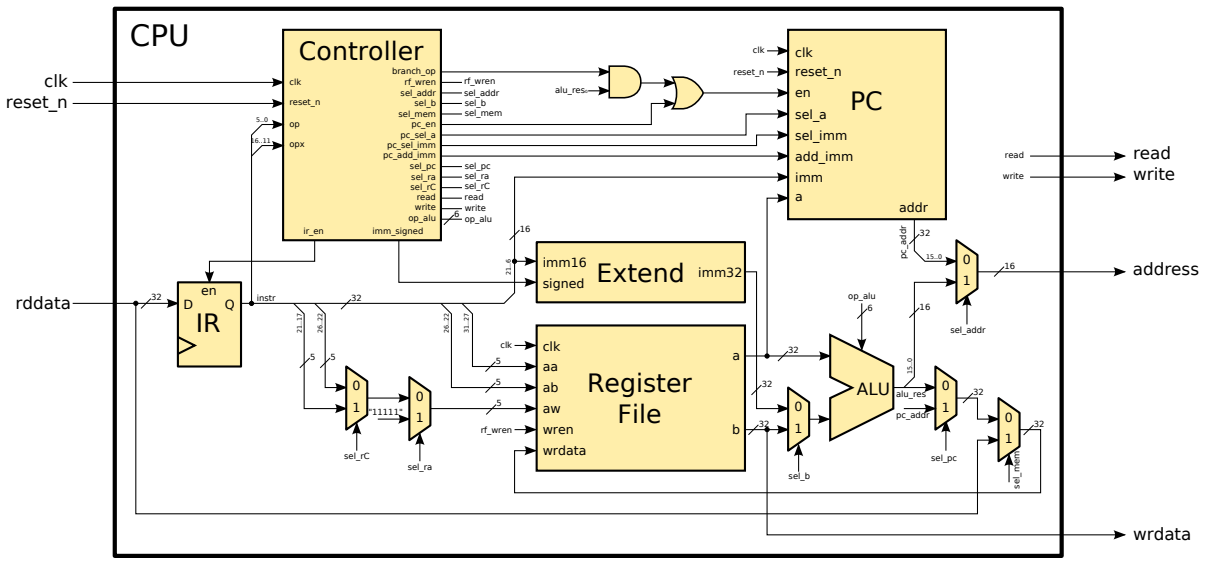
The goal of the first group of labs is to implement a stopwatch on FPGA which can accurately measure the time while reacting to the user's input. To do so, you will first develop the following components:

- 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 (**lab 2**).
- a Nios II processor with interrupt support (**this lab**).

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

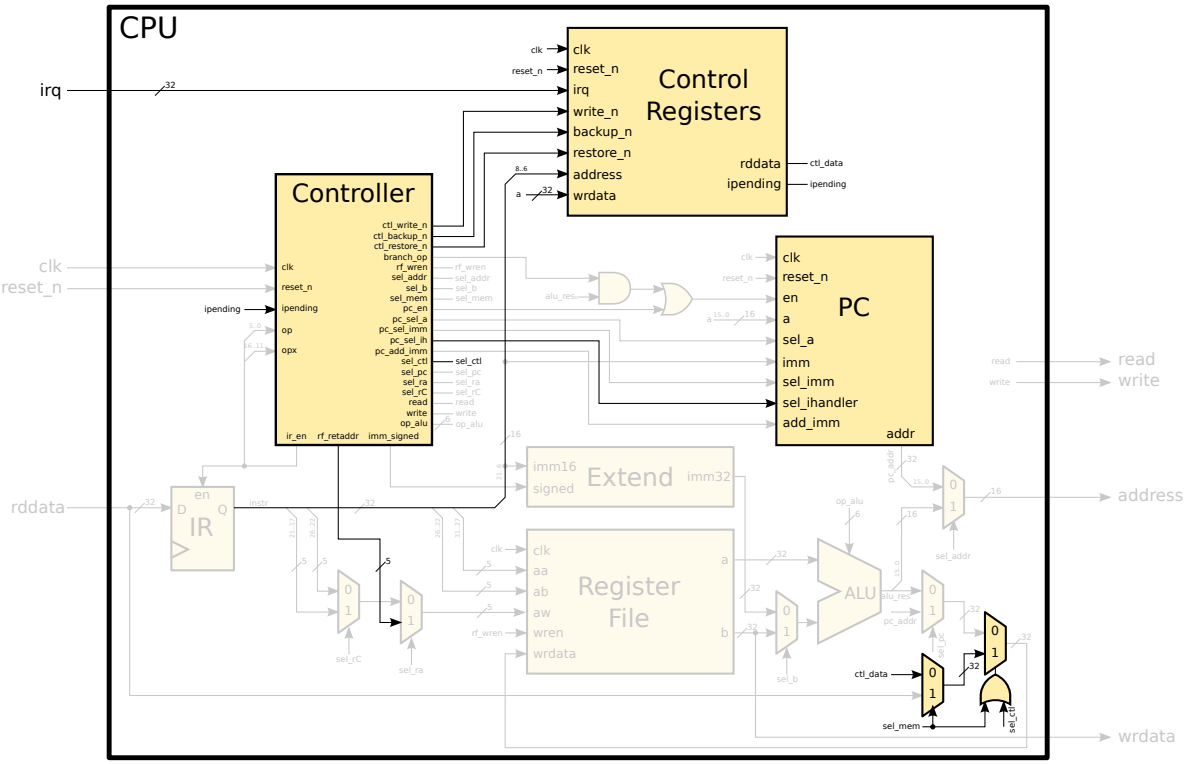
In this lab, you will extend the Nios II processor that you implemented during the ArchOrd course to manage interrupts. For more details on the processor, you can refer to the *Nios II Processor* material available under **Additional material** on the course webpage.

The following schematic represents the architecture of the Nios II that you implemented in ArchOrd.



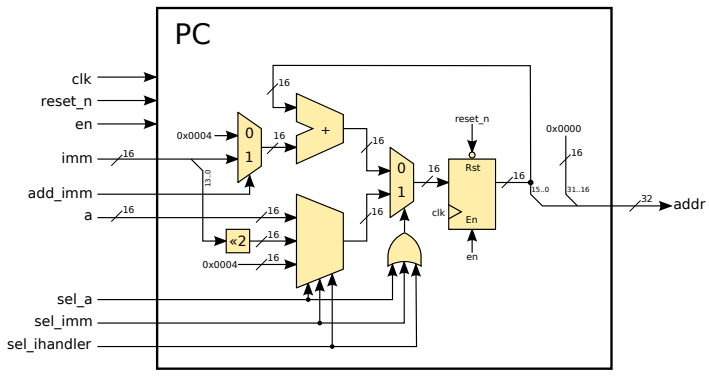
The following figure shows the modifications required for the architecture to handle interrupts. For handling interrupts, a new unit, called **Control Registers**, is added to the architecture. The **irq** input

signal captures the interrupt events. The **Program Counter** and **Controller** need to be modified in order to handle the interrupt events. When an interrupt occurs, the normal program flow should be stopped and the interrupt should be serviced by jumping to the appropriate interrupt service routine. In this experiment, the original architecture is provided and you should apply the required modifications to handle the interrupts.



2 The Program Counter

The **Program Counter** has been slightly modified. It takes an additional **sel_ihandler** input signal. This signal is activated by the **Controller** of the Nios II. When this signal is set, the **Program Counter** provides the address of the interrupt handler.



2.1 Description

The **Program Counter** provides the address of the next instruction to execute. The address is stored in a 16-bit internal counter.

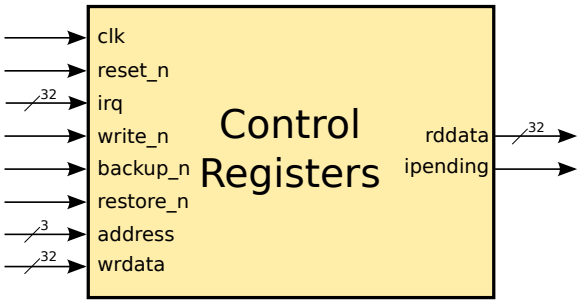
- When **reset_n** is 0, the internal counter is *asynchronously* initialized to zero.
- The **en** input signal enables the internal counter to take a new value, which is selected by the **sel_a**, **sel_imm**, **sel_ihandler** and **add_imm** signals:
  - **sel_a** selects the **a** input.
  - **sel_imm** selects the **imm** input shifted to the left by 2.
  - **sel_ihandler** selects the interrupt handler address (i.e., `0x0004` fixed inside).
  - **add_imm** selects the result of the addition between the **imm** input and the current value of the internal counter.
  - If none of these signals is active, the internal counter is incremented by 4.
  - Activating more than one of these signals will result in an undefined behavior.
- The **addr** output signal is the current value of the internal counter extended to 32 bits.

2.2 Exercise

- Download the project files and open the ModelSim project.
- Open the `PC.vhd` file.
- Implement the **Program Counter** according to its description. An entity is already given which matches the `PC_tb.vhd` testbench.
  - If you have implemented the **Program Counter** during the last semester, you can start from your version and complete it inside the new `PC.vhd` file.
- Run the `PC_tb.vhd` testbench to quickly verify your design.
  - Start a simulation of `PC_tb.vhd`.
  - Add the signals of the **Program Counter** to the wave.
  - Type `run -all` in the command line.
  - A message will tell you whether the basic functionality of your design is correct.

3 The Nios II Control Registers

Interrupts are controlled through the **Control Registers**. By using these registers, you can activate or deactivate interrupts from a particular source and find the interrupts that are currently pending.



- When **reset_n** is 0, every **control register** is *asynchronously* initialized to zero.
- The **irq** input signal is the interrupt source vector.
- When the **write_n** input signal is set to 0, the **wrdata** value is written *synchronously* in the register addressed by the **address** input.
- When the **backup_n** input signal is set to 0, the **PIE** bit in register `status` is saved in the **EPIE** bit in register `estatus` and it is cleared (i.e., set to 0).
- When the **restore_n** input signal is set to 0, the **PIE** bit is restored from the **EPIE** bit.
- Activating more than one of the **write_n**, **backup_n** and **restore_n** input signals at a time will result in an undefined behavior.
- The **address** input signal selects one of the six **control registers**.
- The **wrdata** input signal is the data to write in the **control registers**.
- The **rddata** output signal *asynchronously* reads the **control register** addressed by the **address** signal.
- The **ipending** output signal is set to 1 when the **PIE** bit is 1 and the `ipending` register is not zero.

3.1 Internal Registers

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	

3.1.1 status

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

3.1.2 estatus

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

3.1.3 bstatus

The `bstatus` register is not used for interrupts. (You can simply ignore it).

3.1.4 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* to irq*31*. A value of 1 in bit *n* means that the corresponding irq*n* interrupt is enabled; reversely, a value of 0 means that the corresponding interrupt is disabled.

3.1.5 ipending

`ipending` is not a real register. The value of `ipending` indicates which interrupts are currently pending. It corresponds to the **irq** input signal masked by the `ienable` register. A value of 1 in bit *n* means that the corresponding **irq***N* input is asserted and it is enabled (i.e., `ienable`*n* is 1). The `ipending` register is *read-only* (i.e., writing a value to this register has no effect).

3.1.6 cpuid

The `cpuid` register is not used for interrupts. (You can simply ignore it).

3.2 Exercise

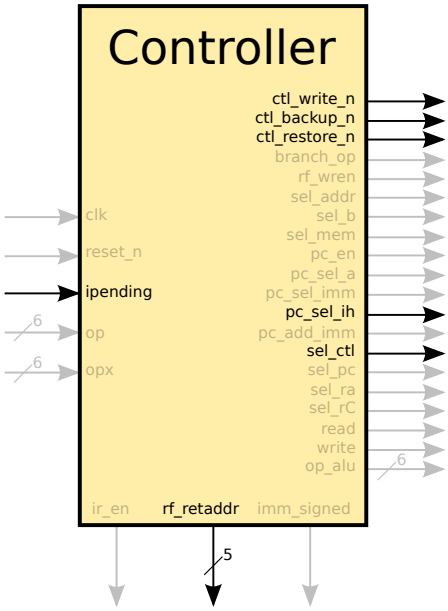
- Open the `control_registers.vhd` file.
- Implement the **control registers** according to its description. An entity is already given which matches the `control_registers_tb.vhd` testbench.
- Run the `control_registers_tb.vhd` testbench to verify your design.
  - Start a simulation of `control_registers_tb.vhd`.
  - Add the signals of the **Control Registers** to the wave.
  - Type `run -all` in the command line.
  - A message will tell you whether the basic functionality of your design is correct.

4 Downloading the Processor onto the FPGA

In the final section of the experiment, you should verify your processor on the Gecko4Education board. You should use all the VHDL files of the processor that you implemented. Before that, the **Controller** should be modified to support interrupts. The **Controller** is given, you do not have to implement it. The new features of the **Controller** are described in the next subsection.

4.1 Description of the Controller

The new input and output signals are highlighted in the following figure.



- The **ipending** input signal comes from the **Control Registers** and tells to the **Controller** if an interrupt is currently pending.

- The **ctl_write_n** output signal is activated during the execution of a `wrctl` instruction to write a value coming from the **Register File** into the **Control Registers**.

- The **sel_ctl** output signal is activated during the execution of a `rdctl` instruction to select the data coming from the **Control Registers** as the data that should be written into the **Register File**.

- The **rf_retaddr** output signal defines a destination register. It is used in case of an interrupt or during a `call` instruction to select the `ea` or `ra` register, respectively.

- The **ctl_backup_n** output signal is activated when the processor jumps to the interrupt handler, to save the **PIE** bit.

- The **ctl_restore_n** output signal is activated during the execution of an `eret` instruction, to restore the **PIE** bit.

- The **pc_sel_ih** output signal is activated when the processor must jump to the interrupt handler.

4.2 Exercise

- Open the Quartus project called `GECKO.qpf` in the **quartus** folder of the template (File >Open Project).

- The project contains a complete multicycle Nios II processor like the one you implemented in ArchOrd. The processor is connected to an interface to the buttons and to the timer that you developed during the *Timer: A Peripheral Interface* lab. If you wish, you can replace each precompiled module of the Nios II with the ones you implemented yourself (except for `controller.vhd`, which is not the same as in ArchOrd) according to the following table:

Compiled file	Source file(s)
ALU.qxp	ALU.bdf add_sub.vhd comparator.vhd logic_unit.vhd shift_unit.vhd multiplexer.vhd
decoder.qxp	decoder.vhd
extend.qxp	extend.vhd
IR.qxp	IR.vhd
mux2x5.qxp	mux2x5.vhd
mux2x16.qxp	mux2x16.vhd
mux2x32.qxp	mux2x32.vhd
register_file.qxp	register_file.vhd

To replace a module, you should first remove the respective source file(s) from the project:

- Make sure that the **Project Navigator** mode is set to **Files** (check the drop down menu to the right of the label Project Navigator).
- Right click on the file(s) to remove and select **Remove File from Project**.

Then, add you source file(s):

- In the Project Navigator, right click on **Files** and then on **Add/Remove Files in Project...**
- Click on **...** next to **File name** and select your file(s).
- If you selected only one file, you should also click on the **Add** button.

- Copy the `timer.vhd` file that you developed during the *Timer: A Peripheral Interface* lab to the **vhdl** folder.
- Add an initialization file for the ROM:
  - In the Nios2Sim simulator, open the program that you implemented during the *Nios II Interruption Simulation* lab.
  - Select File >Export to Hex File..., and select the ROM.
  - Save the program contents in the **quartus** folder in the `ROM.hex` file.
- Compile the Quartus project.
- Program the board.
  - Do you see the same behavior of your program compared to Nios2sim simulation? Why?
- Modify the program to set the period of the timer to 1 second, so that the counter handled by the timer's ISR is incremented each second. The clock frequency of the Gecko4Education board is 50 MHz.
  - Do not forget to export the new program content in the `ROM.hex` file and to recompile the Quartus project before programing the FPGA.
  - Verify that the timer generates one interrupt per second.

5 Submission

Please submit `PC.vhd` and `control_registers.vhd`.