# Interrupt analysis

**DE1-SoC**
Interrupts
**Quartus Prime - NIOSII**

| | |
|---|---|
| **Objectives** | Measuring interrupt parameters on a processor, case study with a Nios II system. Observing latency, response and recovery time on Nios II systems |
| **Tools** | Terasic DE1-SoC, Quartus Prime, Qsys and Nios II SBT |
| **Preliminary** | VHDL, C, Embedded System Architecture, Avalon bus, Nios II processor |
| **Theory** | FPGA, Avalon, Nios II |
| **Material** | Quartus Prime, Terasic DE1-SoC, Saleae Logic Pro 16 Logic Analyzer |
| **Duration** | 3x2h |

## 1 Introduction

The goal of this laboratory is to get in-depth knowledge of the various timings involved in interrupt handling. To achieve this goal, we will measure the delays involved starting from the generation of an interrupt request until the interrupt is completely handled. We will also explore the delays introduced by Real-time operating system (uC/OS-II) synchronization mechanisms.

All measurements will be done on a Cyclone V FPGA. We will use a timer to periodically generate an interrupt when its internal counter reaches 0. When this occurs, a specific function will be called and some **latency** is necessary for the CPU to access the interrupt handler.

The Nios II processor only has one interrupt handler, so a register (`ipending`) must be **polled** to determine the source of the generated interrupt request. Once the device is found, the processor can select the appropriate **Interrupt Service Routine (ISR)** to be called. The time needed to access this function is called the interrupt **response** time. At the end of the ISR, the time needed to jump back to the main program is called the interrupt **recovery** time. From the user's point of view, all of these delays are equivalent to lost CPU time, and therefore to lost performance. The lower the latency, the better the interrupt reactivity of the system is.

In this laboratory, different techniques are used to measure those parameters.

## 2 Design to realize

Please use the (empty) template available on Moodle for this project.

We use an FPGA-based embedded system, thus we need to realize such a system. In this laboratory, you will realize a specific design with the following elements:

- FPGA 5CSEMA5F31C6 on the DE1-SoC board
- Nios II Processor with different configurations. We will **not** use the “Nios II (classic)”, but instead the standard Nios II design.
- Timer IP (from library) for measuring delays
- Performance counter IP (from library) for measuring delays
- SDRAM controller (check SoC-FPGA design guide on Moodle for SDRAM parameters https://moodle.epfl.ch/pluginfile.php/1680499/mod_resource/content/8/SoC-FPGA%20Design%20Guide%20%5BDE1-SoC%20Edition%5D.pdf
- On-chip memory of 128KB, 32 bits width
- JTAG_UART (serial interface)
- Parallel input/output port IP (from library) as **Input** for 4 push button (active low), with interrupt generation
- ***A specific parallel port interface to be designed in VHDL (details below)***
- ***A specific counter interface to be designed in VHDL (details below)***

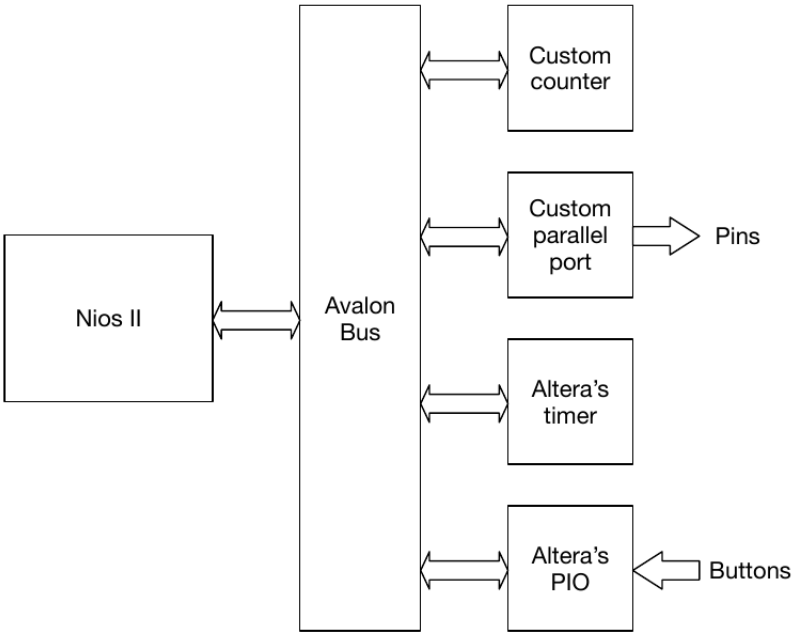


Figure 1: Qsys system I/O (memories not shown)

## 2.1 Specific counter (to design)

You have to realize **your own counter** with the following characteristics:

- 32-bit Avalon slave
- 32-bit counter
- 1 wait cycle for read access (synchronous read)
- Increment the counter at the system's clock speed (50 MHz)
- Command to reset the counter, command to start the counter, command to stop the counter
- The counter value must be readable at all times → transfer the counter value at the start of the read cycle

***Manipulation 1** Counter design*

- Propose a register map of the interface
- Realize and simulate the interface
- Realize a Qsys component from this interface → use the memory model for the Avalon interface

## 2.2 Specific parallel port (to design)

You have to realize a specific parallel port with the following features:

- Generic N-bit parallel port (up to 32 bits)
- 3 supported accesses:
  - *Standard read / write* : Read and write the parallel port value (offset 0)
  - *Set bit*: The bits set to '1' when writing will set the corresponding bits stored in the PIO to **'1'** (offset 1)
  - *Clear bit*: The bits set to '1' when writing will set the corresponding bits stored in the PIO to **'0'** (offset 2)

A single write access can set or clear as many bits as requested. It is not necessary to protect the access in read-modify-write mode (usually done in this case of device if interrupt driven system is used).

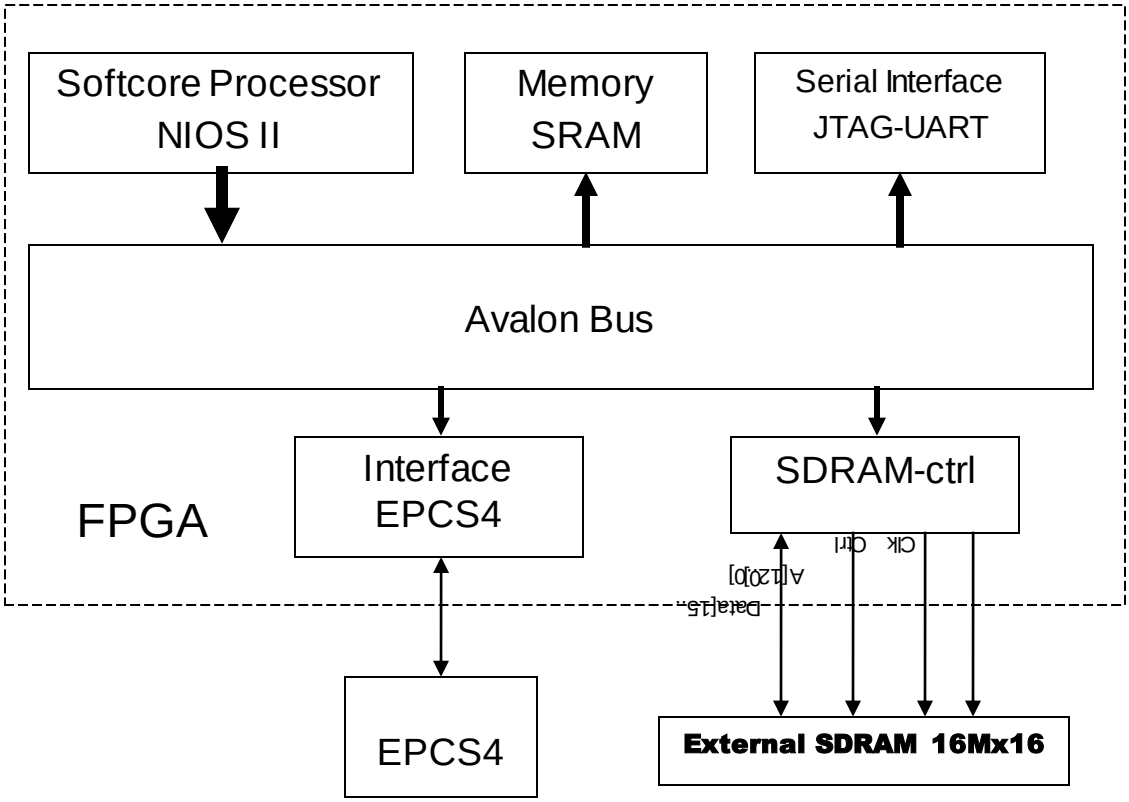


*Figure 1 FPGA, general internal bloc diagram, NOT all functions are included in this figure.* ***EPCS4 is deprecated****.*

### ***Manipulation 2*** *Parallel Port design*

- Propose a register map of the interface
- Realize and simulate the interface
- Realize a Qsys component from this interface → use the memory model for the Avalon interface

# 3 Software design

After the hardware realization, it's time to make software:

Test with the normal timer to determine the response time of an interrupt.

Be careful with the use of "printf()" in interrupt-driven function.

## 3.1 Interruptions test

Some tests for interrupt driven measure.

***Manipulation 3** Interrupt response time*

- Initialize interrupt for the timer
- When the timer reaches 0, it will generate an interrupt (and continue to run!)
- Is it an up or down counter?
- In the corresponding interrupt function, read the timer value after a "snap" access. Thus, it is possible to know the response time. How? You have to answer that.

***Manipulation 4** Interrupt recovery time*

- With the help of your own timer, you will evaluate the interrupt recovery time
- Stop and reset the timer in the main program, before normal timer initialization of manipulation 2.
- Start your timer at the end of your interrupt function
- In the main program wait for timer no zero value read (why?)
- This will represent the response time, how long is it?

## 3.2 Logic Analyzer measurement

We will connect an external logic analyzer (Saleae Logic Pro 16) on the GPIO0 connector of the board.

Do not forget to connect ground between the board and the logic analyzer.

With the logic Analyzer and it's software, and with your own designed parallel port initialized as output, try the following instructions:

- Set bit 0, Clr bit 0, Set bit 0, Clr bit 0 with 4 instructions and loop forever
- Observe the signals with the logic analyzer and make timing measurements with a 50MHz system clock. What is the pulse frequency? How many clock cycles correspond for a parallel port period?
- Try with different processors – memories configurations (6 configurations):
  - Instruction cache **disabled**, data cache **disabled**, on-chip memory
  - Instruction cache **enabled**, data cache **disabled**, on-chip memory
  - Instruction cache **enabled**, data cache **enabled**, on-chip memory
  - Instruction cache **disabled**, data cache **disabled**, sdram memory
  - Instruction cache **enabled**, data cache **disabled**, sdram memory
  - Instruction cache **enabled**, data cache **enabled**, sdram memory

***Manipulation 5** Interruptions measurement*

- Use the same techniques for interrupt latency, response and recovery times by activating I/O bits and clearing them at appropriated times
- Give a small explanation and result of your measures, compare with the timer provided values.
- For the latency time, you have to modify the Altera interrupt handler!

# 4 uC/OS-II

In this part we use the uC/OS-II operating system and measure the overhead of the various synchronization primitives provided by the OS kernel.

## 4.1 ISR

Use the buttons in our design as the interrupt sources to measure the time used by the following uC/OS-II synchronization primitives (detailed below):

- Semaphore
- Flags
- Mailbox
- Queue

In this part of the lab, you need to modify your Qsys system:

- Be sure that the PIO module used for your buttons generated interrupts on ***BOTH*** edges (not only on rising edge or only on falling edge).

## 4.2 Semaphore

Make the main task *wait* on a semaphore. In the ISR for the falling edge of the buttons, *signal* the semaphore to let the main task continue. Measure the overhead of the semaphore.

## 4.3 Flags

Flags allow you to wait on a Boolean condition. For example, you could wait until multiple flags are all enabled (OS_FLAG_WAIT_SET_ALL).

Test the speed of the AND/OR conditions as follows:

- OR: wait on one of the buttons' falling edge.
- AND: wait until the four buttons have been activated.

As always, the *signal* operations must take place in the ISR.

## 4.4 Mailbox

A mailbox is used to transfer a message between tasks. The mailbox in uC/OS-II contains a 32-bit value as payload. If your message fits in the 32-bit field, then you can send it directly through the mailbox. However, if you need to send more than 32 bits of data, you have to put a pointer to your data structure in the mailbox instead.

In the buttons' ISR, send a message through a mailbox. Your message should be a pointer to a structure with 3 members:

- The Buttons number (0..3) pressed
- A boolean value that determines if the interrupt was caused due to a rising or a falling edge of the buttons.
- Time of the event (button press) on 32 bits, with a resolution of 1 us.

## 4.5 Queue

Same as Mailbox, but with Queue

***Manipulation 6***

- Connect the Saleae Logic Pro 16 on the extension connector, with your special PIO output.
- Create the tasks for waiting on the actions of the buttons with the different synchronization mechanisms.
- Each time a *signal* operation is done, activate a bit on the parallel port on the external connector.
- Each time a *wait* operation is done, deactivate the corresponding bit on the external connector.
- Measure the elapsed time with your timer and compare the time captured by the logic analyzer.

# 5 Report

- Provide a report with the components you created/modified in VHDL (PIO and Timer).
- Provide a general schematic of your design from the Buttons to the output port and external connector.
- Provide a table with the measured times obtained by your timer and by the logic analyzer.
- Comment your results.