Lab 2.1

Designing & testing a programmable interface

# Introduction

In the previous lab, you used a system integration tool (Qsys) to create a full FPGA-based system comprised of a processor, on-chip memory, a JTAG UART (the serial port used for standard output), and a simple programmable interface, the parallel input/output (PIO) port.

Until this point you have used pre-existing standard IP components and programmable interfaces from the system integration tool’s library to form your system. However, the primary advantage of using FPGAs is the ability to create custom programmable interfaces that perform specific functionality for your application.

In this lab, we will tackle the problem of designing and testing a custom programmable interface. Once fully designed and debugged, we will proceed to add it to the system integration tool’s library so we can re-use it in future projects.

# Goal

The goal of this lab is to get you acquainted with designing and testing custom programmable interfaces. To this end, we will dive deeper into Altera’s FPGA toolchain and will learn how to:

* Design a custom programmable interface.
* Test the custom peripheral using a testbench that will simulate the component with a simulator.
* Add the custom programmable interface to the system integration tool (Qsys).
* Use our custom peripheral in a full FPGA-based system.

# Theory

The custom programmable interface we are going to develop is the parallel input/output (PIO) port. You used Altera’s standard PIO core in the last lab, but we are now going to design one ourselves in VHDL.

## System Block Diagram

When designing custom logic, the first thing to do is to draw the block diagram of the overall system, and the block diagram of the custom logic that you are implementing.

## Custom Logic Block Diagram

We are designing a slave peripheral that must interface with other peripherals through the Avalon bus. Therefore, we need to know the ports that an Avalon slave peripheral needs to correctly interface with the bus.

Programmable interfaces are circuits that form the basis for specialized functionality in an embedded system. A programmable interface is an instance of a slave which is accessible through a bus by a master, and which the master configures to perform a specific task.

* A master is a device that initiates transactions on a bus,
* A slave is a device that responds to transactions initiated by a master.

One of the simplest programmable interfaces is the PIO (Parallel Input/Output). A PIO is a circuit that provides parallel data input/output and is typically connected to physical pins on an FPGA for either:

* Inputting data from the outside world (e.g. through a set of buttons),
* Outputting data to the outside world (e.g. through a set of LEDs).

For example, a processor (master) can access a PIO (slave) through a bus and configure its direction to “output” mode and its output value to 0x12.

## Designing Programmable Interfaces

Programmable interfaces are

* Written in a Hardware Description Language (VHDL, Verilog)
* Added to a larger system on the FPGA (comprised of processors, memory, …)
* Synthesized by the FPGA tools
* Programmed onto the FPGA

### The Qsys system integration tool

Using a tool is like speaking a language, as you must first learn it before you can use it. The only way to learn a tool is to follow the same method all serious engineers have previously used. It consists of an ancestral technique that can be summarized by 1 acronym: RTFM. If you are not familiar with this acronym, we highly suggest you look it up and add it to your skillset. It will save you numerous times in this industry.

Download the [Quartus Prime Standard Edition Handbook](https://www.altera.com/en_US/pdfs/literature/hb/qts/qts-qps-handbook.pdf) and read the chapters relevant to Qsys. For this first introduction to Qsys, it is enough to only read “Chapter 5: Creating a System With Qsys”.

You don’t need to read the full chapters, but reading through at least the following sections gives you the big picture of what Qsys can do.

* Volume 1 – Chapter 5: Creating a System With Qsys
  + Create a Qsys System (pg 189 – 200)
  + Integrate a Qsys System and the Quartus Prime Software With the .qsys File (pg 237)

### The Avalon Bus

When you add IP components to a system and connect them to each other, Qsys generates a bus through which all the components can communicate. Many bus designs exist in the industry, and Altera FPGAs use the bus called the Avalon bus. You will learn more about the details of the Avalon bus in the course, but what we need to be concerned with at this stage is how masters “see” slaves through this bus.

The Avalon bus uses memory-mapped I/O. This means that the same address bus is used to access memory and I/O devices. With this information we can now say that masters “see” slaves at specific addresses in the master’s address space. For example, suppose that a peripheral is visible at address 0x1000 in a master’s address space, then the master can read/write to the interface by reading/writing at address 0x1000.

# Practice – Hardware

## Prerequisites

1. Download the provided basic project directory [template](http://moodle.epfl.ch/mod/resource/view.php?id=924083) from Moodle.
2. Decompress the archive.
3. Rename the extracted archive to “fpga\_design\_and\_testing”.
4. Read the README file contained in the project directory before continuing.

## Creating a Quartus Prime Project

1. Go to **File->New Project Wizard…**
   1. Choose “fpga\_design\_and\_testing/hw/quartus” as the working directory.
   2. Use “fpga\_intro” as the project name.
   3. Click the **Finish**.
2. We are using the DE0-Nano-SoC board, so we are going to execute a script to configure the FPGA family and the pin assignments in Quartus Prime.
3. Go to **Tools->Tcl Scripts…**
4. Select “pin\_assignment\_DE0\_Nano\_SoC.tcl”.
5. Click **Run** and be patient (Quartus Prime might freeze for a while).
6. Then, we need to add the top-level entity of the project so Quartus Prime can have a starting point to compile the design.
7. Go to **Project->Add/Remove Files in Project…**
8. Click on the button labelled **…**
9. Select “fpga\_intro/hw/hdl/DE0\_Nano\_SoC\_top\_level.vhd”and click **Open**.
10. Click **Add**. Note that the added path is relative from the Quartus Prime project directory, i.e. “../hdl/DE0\_Nano\_SoC\_top\_level.vhd”.
11. Click **Ok.**
12. In the **Project navigator**, switch to the **Files** view.
13. Right click on “../hdl/DE0\_Nano\_SoC\_top\_level.vhd” and click on **Set as Top-Level Entity**.

## Creating a Qsys System

We will create a full system including a processor and memory capable of executing software that we are going to write in C.

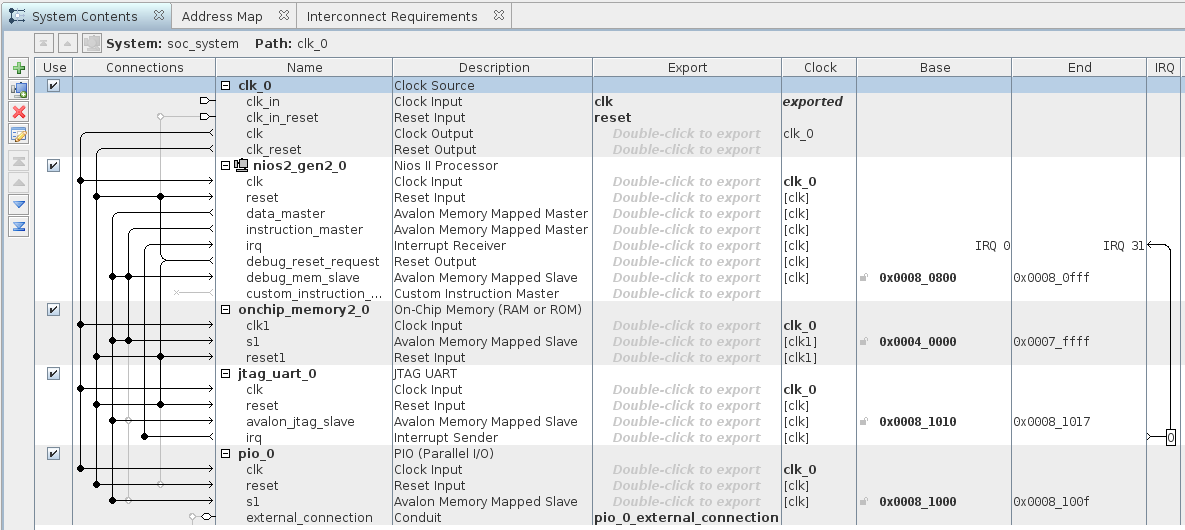
1. Launch Qsys by going to **Tools->Qsys**.
2. In Qsys, go to **File->Save as** and save the Qsys system as “system.qsys” under “fpga\_intro/hw/quartus”.
3. Use the **IP Catalog** to add the following components:
   1. A “Nios II Processor” to act as our main processor.
      1. Click **Finish** on the configuration view that just opened.
   2. An “On-Chip Memory (RAM or ROM)” to act as our main memory.
      1. Choose its **Total memory size** to be 128k and hit the tab key on your keyboard to autocomplete.
      2. Click **Finish** on the configuration view.
   3. A “JTAG UART” to see the results of the standard output on your computer screen.
      1. Click **Finish** on the configuration view that just opened.
   4. A “PIO (Parallel I/O)” to toggle LEDs on the board.
      1. Use a **Width** of 8 bits because the DE0-Nano-SoC has 8 LEDs.
      2. Select **Output** as the direction.
      3. Click **Finish** on the configuration view.
   5. Now it’s time to connect the system components together. Here are a few tips to do that correctly:
      1. Go to **System->Create Global Reset Network** to stop thinking about manually connecting the reset *everywhere*.
      2. Connect the “clk” interface of the “clk\_0” component to the “Clock input” interface of all the other components.
      3. Connect the instruction bus (“instruction\_master”)of the CPU to the on-chip memory.
      4. Connect the data bus (“data\_master”) of the CPU to all the other components.
      5. Connect the Interrupt Sender interface of the JTAG UART to the Interrupt Receiver interface of the CPU.
   6. We need to export the“external\_connection” of “pio\_0” by double-clicking in the **Export** column and pushing ↵ on your keyboard. Recall: exporting a signal means making it available outside the Qsys system to route it on the board.
   7. Double-click on the “nios2\_gen2\_0” component to open up its configuration view.
      1. Under the **Vectors** tab, select the on-chip memory to be its **Reset vector memory** and **Exception vector memory**.
      2. The **Reset vector offset** is the offset from the base address of the on-chip memory to which the CPU jumps upon reset.
      3. Similarly, the **Exception vector offset** is the offset to which the CPU jumps upon receiving hardware exceptions or traps.
      4. You can now close the configuration view.
   8. Go to **System->Assign Base Addresses** to properly configure the address range of each component and avoid conflicts.
   9. Similarly, go to **System->Assign Interrupt Numbers**.
   10. Go to **File->Save**.
4. Your system should now look like the following – don’t worry if you have different Base addresses than in this example:  
   

Figure . Qsys System

1. You can now close Qsys by clicking on **Finish**. Do not generate the system when asked.
2. Use the same procedure you used to add the top-level VHDL file to the project to, this time, add the “fpga\_intro/hw/quartus/system.qsys” file to the project.
3. You should now update your top-level to include your Qsys system.
4. In a text editor, open “fpga\_intro/hw/quartus/system/system\_inst.vhd” to get the VHDL component declaration and instantiation template of the Qsys system.
5. Add the component declaration to your top-level architecture.
6. Add the component instantiation and map the port as follows:
   * 1. The clock should be routed to “FPGA\_CLK1\_50”, the clock input pin of the FPGA/SoC device.
     2. Use the “KEY\_N(0)” button as the reset signal of your design.
     3. Route the PIO to “LED”, the pins connected to the LEDs on the board.
7. Then, in the entity, comment all the ports you are not using, i.e. everything except the 3 mentioned above. Be careful with your semi-colons!
8. Go to **Processing->Start Compilation** and grab a coffee/tea or even a pineapple juice, we are open-minded.
9. Once the compilation is finished, plug in your FPGA board and go to **Tools->Programmer**.
10. Click **Auto-detect** and select **5CSEMA4**.
11. Right-click on the beautiful picture of Altera chip labelled **5CSEMA4** and select **Change File…**
12. Select “fpga\_intro/hw/quartus/output\_files/fpga\_intro.sof”*.*
13. Enable the “Program/Configure” checkbox for device **5CSEMA4U23**.
14. Press **Start**.

# Practice – Software

## Creating a Nios II SBT Project

1. Launch a **Nios II Command Shell** from the start menu of your Windows machine.
2. Use the following command to launch the IDE: “eclipse-nios2 &”.
3. Go to **File->New->Nios II Application and BSP from Template**.
4. Select “fpga\_intro/hw/quartus/system.sopcinfo” as **SOPC Information File name**.
5. Name your software project “fpga\_intro”.
6. We invite you to uncheck the **Use default location** checkbox and choose  
   “fpga\_intro/sw/nios/application”. We encourage this practice to properly separate software from hardware design files.
7. Choose **Hello World** as the **Project template**.
8. Click **Finish**.

## Programming the Nios II Processor

At this stage you can start writing a C program that once built can be run on the Nios II processor that we previously instantiated on the FPGA. The goal is to access and configure the programmable interface that we added to the system, the PIO port.

However, in order to use the PIO programmable interface, we need to know its interface. This interface has multiple names in the community, but the general consensus is that the interface is called the component’s register map. Since you did not design the PIO peripheral yourselves, you do not know its register map. So we need to look into the peripheral’s [documentation](https://www.altera.com/en_US/pdfs/literature/ug/ug_embedded_ip.pdf) to see it.

Chapter 11 describes the PIO core. Good practice says that one should read the full documentation of the IP cores that they use. In particular, we are interested in the subsection called “Software Programming Model” (pg 141-144) which shows the peripheral’s register map along with a detailed description of each register.

The documentation also mentions that the core comes with a header file “altera\_avalon\_pio\_regs.h” that defines the core’s register map. You can include this header in your code so you can use symbolic constants (macros) to access low-level hardware instead of hard-coding various numbers in your programs.

Please avoid hard-code constants in your code and instead use named macros whenever possible. It makes the code much easier to read, which helps US help YOU when you have problems. This comment is equally important in industry as well, not just at university.

### Performing IO with Peripherals

In order to read and write to system peripherals, you have to include the “io.h” header file in your source code. This header file defines macros which compile into special processor instructions targeted at peripheral IO. These instructions are special because they are part of the IO family of load and store instructions and bypass all caches. The available instructions are listed below.

* Reading
  + IORD\_8DIRECT(BASE, OFFSET)
  + IORD\_16DIRECT(BASE, OFFSET)
  + IORD\_32DIRECT(BASE, OFFSET)
* Writing
  + IOWR\_8DIRECT(BASE, OFFSET, DATA)
  + IOWR\_16DIRECT(BASE, OFFSET, DATA)
  + IOWR\_32DIRECT(BASE, OFFSET, DATA)

### Addressing Peripherals

The macros defined above contain a BASE parameter. This is the base address of the peripheral as defined by Qsys in Figure 2. You can find the base addresses of all peripherals connected to the Nios II processor by including the “system.h” header file in your source code.

With all this in mind, the example below shows how one can perform a simple read operation from the PIO peripheral’s data register.

#include <inttypes.h>  
#include "system.h"  
#include “io.h”  
#include “altera\_avalon\_pio\_regs.h”

int main(void) {  
 uint32\_t pio\_data = IORD\_ALTERA\_AVALON\_PIO\_DATA(base)

return 0;

}

Try to do write a loop where you generate a moving “1” on the LEDs. You should periodically write the following bit patterns to the PIO core:

“00000001” 🡪 “00000010” 🡪 … 🡪 “01000000” 🡪 “10000000” 🡪 “00000001” 🡪 …