Lab 2.0
Introduction to FPGA development

Introduction

An FPGA (Field Programmable Gate Array) is a device capable of programmatically implementing digital circuits at *runtime* (instead of at *compile-time* with ASICs). FPGAs are typically built with elementary units called look-up tables (LUTs), which are programmable logic circuits that can be configured to produce any logic function. Millions of LUTs can be found in today's large FPGAs, therefore allowing complex circuits to be implemented.

In this Embedded Systems course, we will learn to use FPGAs for developing, testing, and implementing application-specific digital circuits and accelerators through a series of labs. At the end of the course, *you* will implement a system comprised of a camera and an LCD controller where pictures can be taken with a camera and displayed on an LCD. Stay tuned to learn more!

Goal

The goal of this lab is to get you acquainted with Altera's FPGA toolchain and for you to get a solid understanding of *programmable interfaces*. To this end, you will:

- Create a full FPGA-based system (containing a processor, memory, and a programmable interface)
- Write a simple C program to control the various components of the system, most importantly the programmable interface.

We will *not* be designing any programmable interfaces in this lab as the goal here is to get familiar with the tools and the environment. We will tackle the design of programmable interfaces in the next lab.

René Beuchat, Sahand Kashani

Theory

Programmable Interfaces

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

Designing Systems

In the previous section, we said that programmable interfaces are written in a HDL, and are then added to a larger *system*, but what is a system?

A system is a set of interconnected components that can collectively be used to implement some functionality. Basic functionality in a system can be obtained by a processor alone, and specialized programmable interfaces can be added to enhance the system's abilities.

Creating a system manually

Since we are working with FPGAs, it is possible to implement all system components directly in VHDL. As a matter of fact, at EPFL's computer architecture course, students are guided in a step-by-step process and build the full system shown in Figure 1 *manually*. This is done by implementing each system component (processor, memory, LEDs, buttons, timer …) from scratch in VHDL, and by connecting them all together through a bus. The problem with this design is that all components are custom-made and cannot be used in another system without significant modifications. Additionally, the design suffers from low performance due to its very simple microarchitecture.

Manually creating systems is great for *learning* how computer systems work and interact, since one starts from the basics and builds the system incrementally. However, such systems are unsuitable for production environments where high performance is expected and is the norm.

All systems use a standard set of components (processors, memories, timers …), so one could theoretically spend a considerable amount of time to create such components for re-use in various projects. However, the amount of time required to implement and debug such designs would greatly surpass the time available for the project you have been hired for! *There must be a way for engineering time to be better spent.*

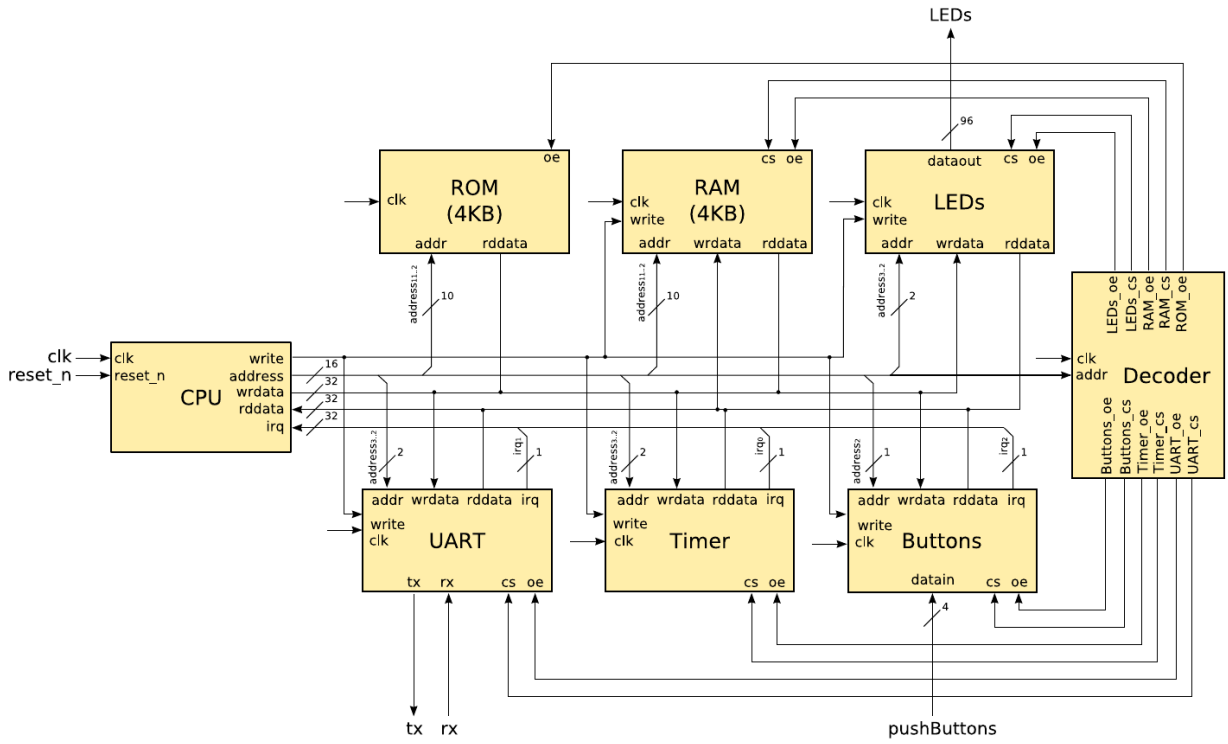

FIGURE 1. CUSTOM-MADE SYSTEM COMPONENTS INTERCONNECTED WITH A SHARED BUS.

Creating a system automatically

Solutions exist for the previously described problem, and come in the form of *system integration tools*. Most FPGA & ASIC design tools have their own system integration tool, but they all essentially expose the same functionality.

A system integration tool provides a *library* of *standard components*. Designers can choose any component from the library and use the tool to connect them together. All components don't forcefully have the same interface, but most tools generally take care of this by inserting adapters to convert between the different bus sizes and handle minor timing adjustments required by each component.

Using such a tool, we can connect *standard interfaces* together to build up the *backbone* of the system, and concentrate on implementing *custom hardware* only for our *specialized* task.

We will look at one such tool (Qsys) provided by Altera for use in their FPGAs.

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 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 from Moodle.
2. Decompress the archive.
3. Rename the extracted archive to "`fpga_intro`".
4. *Read the README file* contained in the project directory before continuing.

Creating a Quartus Prime Project

5. Go to **`File->New Project Wizard…`**
   a. Choose "`fpga_intro/hw/quartus`" as the working directory.
   b. Use "`fpga_intro`" as the project name.
   c. Click the **`Finish`**.

6. 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.
   a. Go to **`Tools->Tcl Scripts…`**
   b. Select "`pin_assignment_DE0_Nano_SoC.tcl`".
   c. Click **`Run`** and *be patient* (Quartus Prime might freeze for a while).

7. Then, we need to add the top-level entity of the project so Quartus Prime can have a starting point to compile the design.
   a. Go to **`Project->Add/Remove Files in Project…`**
   b. Click on the button labelled …
   c. Select "`fpga_intro/hw/hdl/DE0_Nano_SoC_top_level.vhd`" and click **`Open`**.
   d. Click **`Add`**. Note that the added path is relative from the Quartus Prime project directory, i.e. "`../hdl/DE0_Nano_SoC_top_level.vhd`".
   e. Click **`Ok.`**
   f. In the **`Project Navigator`**, switch to the **`Files`** view.
   g. 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.

8. Launch Qsys by going to **`Tools->Qsys`**.

9. In Qsys, go to **`File->Save as`** and save the Qsys system as "`system.qsys`" under "`fpga_intro/hw/quartus`".

10. Use the **`IP Catalog`** to add the following components:
    a. A "`Nios II Processor`" to act as our main processor.
       i. Click **`Finish`** on the configuration view that just opened.
    b. An "`On-Chip Memory (RAM or ROM)`" to act as our main memory.
       i. Choose its **`Total memory size`** to be 128k and hit the tab key on your keyboard to autocomplete.
       ii. Click **`Finish`** on the configuration view.
    c. A "`JTAG UART`" to see the results of the standard output on your computer screen.

   i. Click **Finish** on the configuration view that just opened.

d. A "PIO (Parallel I/O)" to toggle LEDs on the board.
   i. Use a **Width** of 8 bits because the DE0-Nano-SoC has 8 LEDs.
   ii. Select **Output** as the direction.
   iii. Click **Finish** on the configuration view.
e. Now it's time to connect the system components together. Here are a few tips to do that correctly:
   i. Go to **System->Create Global Reset Network** to stop thinking about manually connecting the reset *everywhere*.
   ii. Connect the "clk" interface of the "clk_0" component to the "Clock input" interface of all the other components.
   iii. Connect the instruction bus ("instruction_master") of the CPU to the on-chip memory.
   iv. Connect the data bus ("data_master") of the CPU to all the other components.
   v. Connect the Interrupt Sender interface of the JTAG UART to the Interrupt Receiver interface of the CPU.
f. 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.
g. Double-click on the "nios2_gen2_0" component to open up its configuration view.
   i. Under the **Vectors** tab, select the on-chip memory to be its **Reset vector memory** and **Exception vector memory**.
   ii. The **Reset vector offset** is the offset from the base address of the on-chip memory to which the CPU jumps upon reset.
   iii. Similarly, the **Exception vector offset** is the offset to which the CPU jumps upon receiving hardware exceptions or traps.
   iv. You can now close the configuration view.
h. Go to **System->Assign Base Addresses** to properly configure the address range of each component and avoid conflicts.
i. Similarly, go to **System->Assign Interrupt Numbers**.
j. Go to **File->Save**.

11. Your system should now look like the one shown in Figure 2 – don't worry if you have different base addresses than in the example.

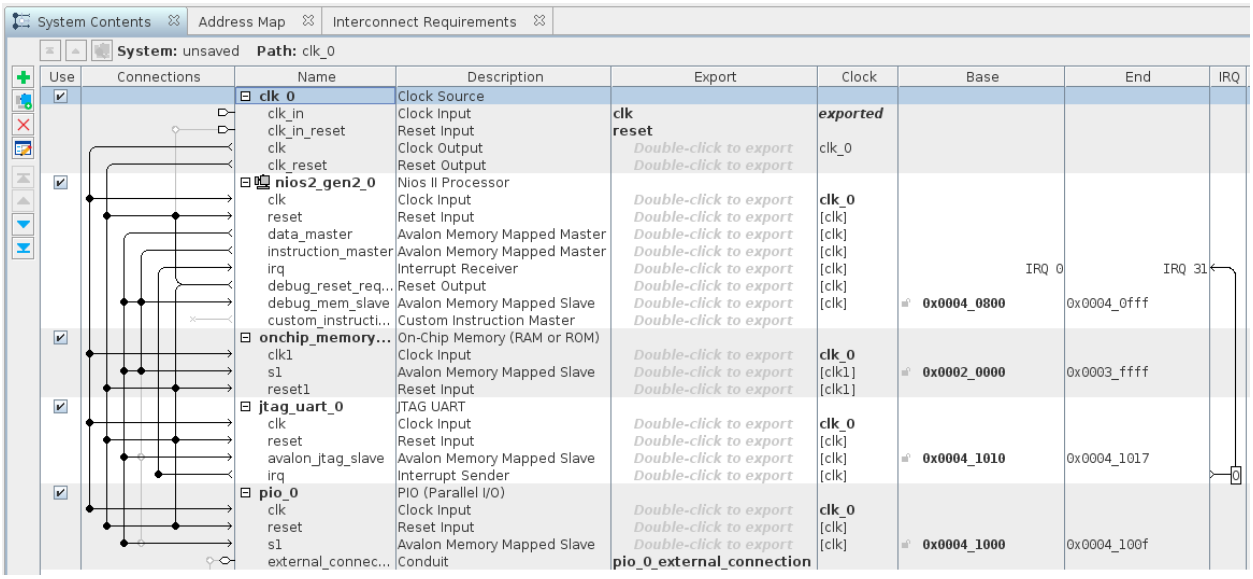

FIGURE 2. QSYS SYSTEM

12. You can now close Qsys by clicking on **`Finish`**. Do *not* generate the system when asked.

13. 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.

14. You should now update your top-level to include your Qsys system.
    a. 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.
    b. Add the component declaration to your top-level architecture.
    c. Add the component instantiation and map the port as follows:
        i. The clock should be routed to "`FPGA_CLK1_50`", the clock input pin of the FPGA/SoC device.
        ii. Use the "`KEY_N(0)`" button as the reset signal of your design.
        iii. Route the PIO to "`LED`", the pins connected to the LEDs on the board.
    d. 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!

15. Go to **`Processing->Start Compilation`** and grab a coffee/tea or even a pineapple juice, we are open-minded.

16. Once the compilation is finished, plug in your FPGA board and go to **`Tools->Programmer`**.
    a. Click **`Auto-detect`** and select **`5CSEMA4`**.
    b. Right-click on the beautiful picture of Altera chip labelled **`5CSEMA4`** and select **`Change File…`**
    c. Select "`fpga_intro/hw/quartus/output_files/fpga_intro.sof`".
    d. Enable the "`Program/Configure`" checkbox for device **`5CSEMA4U23`**.
    e. Press **`Start`**.

Practice – Software

Creating a Nios II SBT Project

17. Launch a **`Nios II Command Shell`** from the start menu of your Windows machine.

18. Use the following command to launch the IDE: "`eclipse-nios2 &`".

19. Go to **`File->New->Nios II Application and BSP from Template`**.

20. Select "`fpga_intro/hw/quartus/system.sopcinfo`" as **`SOPC Information File name`**.

21. Name your software project "`fpga_intro`".

22. 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.

23. Choose **`Hello World`** as the **`Project template`**.

24. 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 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 do not 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" → …