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.

René Beuchat, Sahand Kashani

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 so we have a clear view of where our programmable interface is located in the system. Starting to implement a programmable interface without knowing how it interfaces with the rest of the system is a recipe for disaster as there is a high chance your programmable interface will not be able to connect to the other units in the system.

In this lab we are going to implement a parallel port which interfaces with a processor (the Nios II CPU) through the *Avalon bus*. Figure 1 gives a high-level overview of our target system.

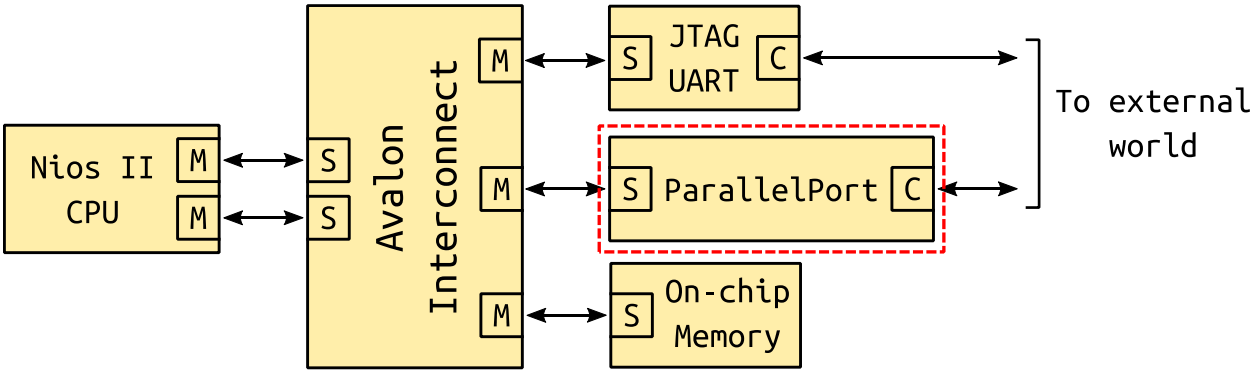

FIGURE 1. HIGH-LEVEL BLOCK DIAGRAM OF THE SYSTEM

Creating a Custom Programmable Interface in VHDL

A custom programmable interface can only interact with the CPU if both entities share a common *interface*. In Intel FPGAs this interface is the *Avalon bus* and its full specification can be found in the Avalon Interface Specifications. Specifications are quite verbose as they define a large number of interface signals and it is sometimes difficult to know how to start. However, since we are designing a very simple programmable interface in this lab, we can restrict ourselves to a small subset of the Avalon interface signals.

The most typical programmable interfaces look like Figure 2 and have a *slave* config interface and a peripheral-specific *conduit* interface[1]. A slave interface is a standard interface that is supposed to connect to a master interface and its signals are therefore defined by the Avalon specifications. A conduit, however, is a collection of signals that are supposed to exit the system and connect with the non-Avalon world. For example, a conduit could correspond to the RX/TX pins of a UART peripheral, or simply to the physical pin connection for a PIO component.

[1] "Conduit" is an Avalon-specific word. Other busses often simply call such collections "external interfaces".

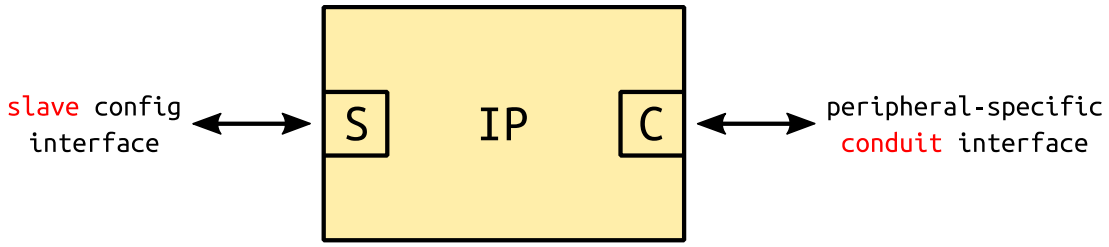

FIGURE 2. AVALON-MM GENERIC SLAVE IP CORE

Now that we know what a typical slave should look like, let's see how to implement our custom PIO interface. Figure 3 dives into the high-level slave and conduit interface shown earlier and exposes the signals they represent under the hood.

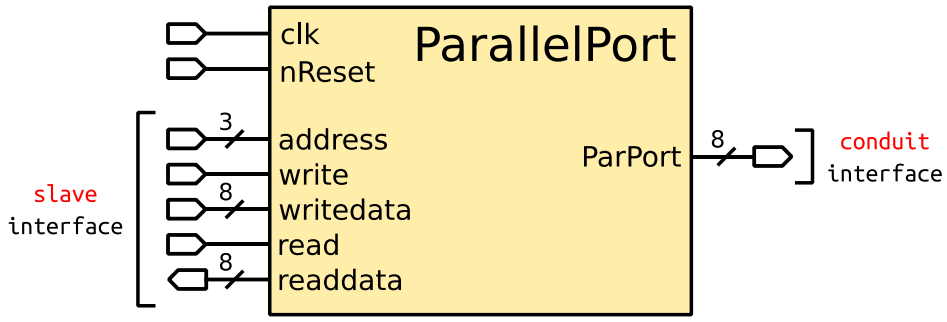

FIGURE 3. PIO COMPONENT INTERFACE

A PIO core is a programmable interface and it must therefore be, uh, programmable. Being programmable means that the component exposes a *register interface*, i.e. a set of registers through which a master can control the functionality of the component. We are going to use the same register interface described in the course for our custom PIO component, namely:

1. `RegDir` – A location in the register interface where the bit-level directionality of the PIO unit can be controlled.
2. `RegPin` – A location in the register interface where the pin-level state of the port can be read if the direction of the PIO unit is set to "input".
3. `RegPort` – A location in the register interface that memorizes a value to be outputted by the PIO unit on its physical pins if the direction of the PIO unit is set to "output".
4. `RegSet` – A location in the register interface to which a write to a specific bit automatically sets the corresponding bit in `RegPort` to 1. All other bits of `RegPort` remain unchanged.
5. `RegClr` – A location in the register interface to which a write to a specific bit automatically clears the corresponding bit in `RegPort` to 0. All other bits of `RegPort` remain unchanged.

Since there are 5 registers in our register map, the smallest address bus width we could use to index all registers is 3 bits. This allows us to index in the range 0-7, even though addresses 5-7 will be unused. This is not a problem though as unused entries in the register interface could be used for future expansion if additional functionality could be needed for the PIO component. Table 1 summarizes the write and read behavior of these registers.

Address	Write register	Writedata[7..0]	Read register	Readdata[7..0]
0	RegDir	→ iRegDir	RegDir	iRegDir →
1	-	Don't care	RegPin	ParPort →
2	RegPort	→ iRegPort	RegPort	iRegPort →
3	RegSet	→ iRegPort	-	0x00
4	RegClr	→ iRegPort	-	0x00
5	-	Don't care	-	0x00
6	-	Don't care	-	0x00
7	-	Don't care	-	0x00

TABLE 1. REGISTER MAP OF THE CUSTOM PIO COMPONENT

Finally, Figure 4 ends with a schematic of the core logic needed to implement the functionality of a parallel port, i.e. the conduit-side behavior of the component.

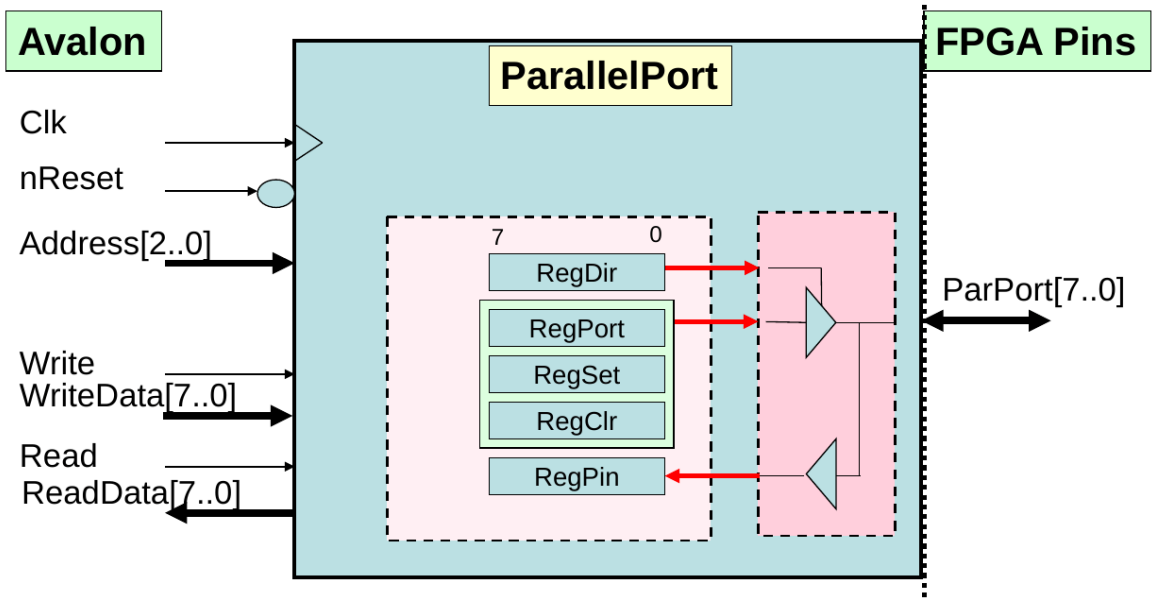

FIGURE 4. CUSTOM PIO COMPONENT IMPLEMENTATION

All that's left at this stage is to implement the Avalon-side of the component with its 0-wait state write transfers and its 1-wait state read transfers. The listing below shows how this is done in VHDL.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ParallelPort is
  port(
    clk    : in std_logic;
    nReset : in std_logic;

    -- Internal interface (i.e. Avalon slave).
    address   : in  std_logic_vector(2 downto 0);
    write     : in  std_logic;
    read      : in  std_logic;
```

```
    writedata : in  std_logic_vector(7 downto 0);
    readdata  : out std_logic_vector(7 downto 0);

    -- External interface (i.e. conduit).
    ParPort : inout std_logic_vector(7 downto 0)
  );
end ParallelPort;

architecture comp of ParallelPort is

  signal iRegDir  : std_logic_vector(7 downto 0);
  signal iRegPort : std_logic_vector(7 downto 0);
  signal iRegPin  : std_logic_vector(7 downto 0);

begin

  -- Parallel Port output value.
  process(iRegDir, iRegPort)
  begin
    for i in 0 to 7 loop
      if iRegDir(i) = '1' then
        ParPort(i) <= iRegPort(i);
      else
        ParPort(i) <= 'Z';
      end if;
    end loop;
  end process;

  -- Parallel Port input value.
  iRegPin <= ParPort;

  -- Avalon slave write to registers.
  process(clk, nReset)
  begin
    if nReset = '0' then
      iRegDir  <= (others => '0');
      iRegPort <= (others => '0');
    elsif rising_edge(clk) then
      if write = '1' then
        case Address is
          when "000"  => iRegDir  <= writedata;
          when "010"  => iRegPort <= writedata;
          when "011"  => iRegPort <= iRegPort or writedata;
          when "100"  => iRegPort <= iRegPort and not writedata;
          when others => null;
        end case;
      end if;
    end if;
  end process;

  -- Avalon slave read from registers.
  process(clk)
```

```
  begin
    if rising_edge(clk) then
      readdata <= (others => '0');
      if read = '1' then
        case address is
          when "000"  => readdata <= iRegDir;
          when "001"  => readdata <= iRegPin;
          when "010"  => readdata <= iRegPort;
          when others => null;
        end case;
      end if;
    end if;
  end process;

end comp;
```

Testing the custom parallel port

An engineer's job does not stop after having created a "solution" to a specific problem, but he/she must also be able to demonstrate, to various degrees of certitude, that the solution is *correct*. This statement is equally valid in both software & hardware engineering, but it is especially important in the hardware domain, as errors can generally not be fixed once a product has been shipped to customers!

When describing digital circuits in VHDL, one generally tests the correctness of their implementation with a VHDL *testbench*. A testbench is generally a *non-synthesizable* VHDL file which iteratively applies a sequence of controlled inputs to a circuit and compares its concrete output against the expected output. If a mismatch is detected, an error is displayed in the VHDL simulator's log which can then be consulted to help direct a designer search for the problem in the circuit's RTL description.

On the moodle page of the course you will find two tutorials for testing VHDL designs using (1) a manually-checked testbench, and (2) a self-checking testbench. We encourage you to read these tutorials if you are not familiar with writing testbenches. These tutorials also introduce *ModelSim*, one of the industry-standard simulators used for simulating hardware designs.

Integrating the custom parallel port with Qsys

Having the VHDL description of the custom parallel port is not enough for the unit to be used in Qsys. Indeed, you need to make Qsys understand how your VHDL design maps to signals in the Avalon bus. This is done through the Qsys GUI by labelling the different ports of the VHDL design's entity with the corresponding Avalon signal category to which it belongs. So, you basically need to get your hands dirty and understand how Qsys works in order to integrate your custom parallel port in the overall design.

Download the Quartus Prime Standard Edition Handbook and read the chapters relevant to Qsys. For this second introduction to Qsys you'll need to read "Chapter 5: Creating a System With Qsys" and "Chapter 6: Creating Qsys Components". You don't need to read the full chapters, but reading through the following sections will teach you how to integrate your VHDL parallel port into Qsys.

- Volume 1 – Chapter 5: Creating a System With Qsys (pg 179)

- Interface Support in Qsys (pg 180)
- Adding IP Cores to the IP Catalog (pg 181)
- Set up the IP Index File (.ipx) to Search for IP Components (pg 184)
- Create a Qsys System (pg 185 – 208)
- Integrate a Qsys System and the Quartus Prime Software With the .qsys File (pg 233)
- View the Qsys HDL Example (pg 251)

- Volume 1 – Chapter 6: Creating Qsys Components (pg 362)
  - Qsys Components (pg 362)
  - Create IP Components in the Qsys Component Editor (pg 366 – 368)
  - Specify IP Component Type Information (pg 368)
  - Specify HDL Files for Synthesis in the Qsys Component Editor (pg 373)
  - Analyze Synthesis Files in the Qsys Component Editor (pg 374)
  - Add Signals and Interfaces in the Qsys Component Editor (pg 378)

Practice

For this lab we recommend you make a 2nd copy of the system in Figure 5 that you built in lab 2.0. This system used the PIO library component and your job is to replace it with the custom ParallelPort component we defined earlier.

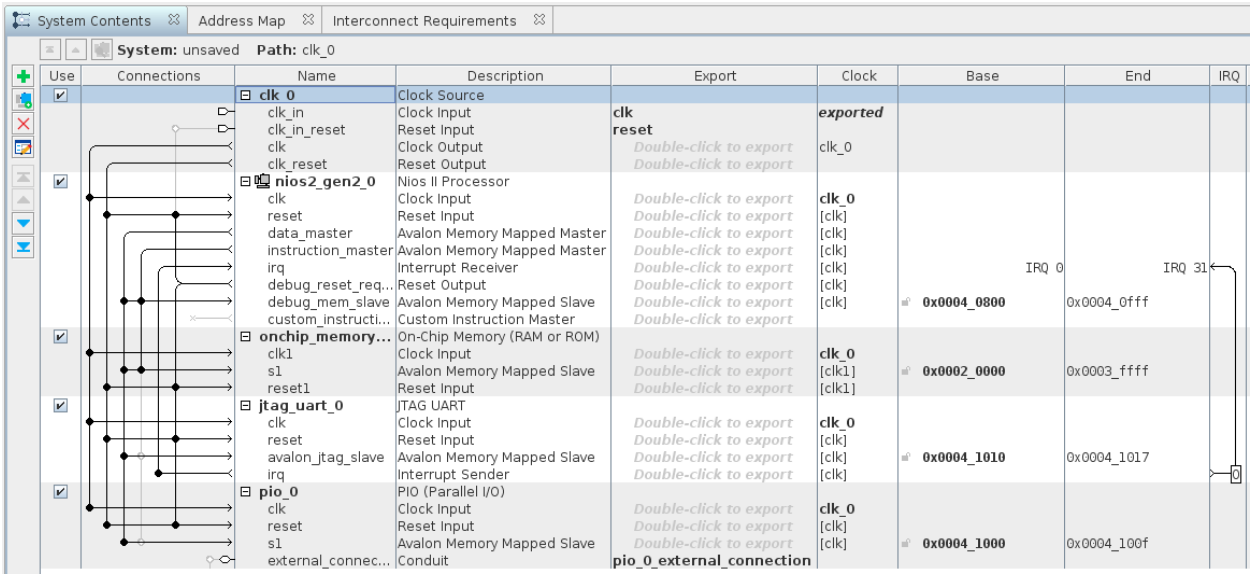

Figure 5. Lab 2.0 Qsys system

1. Delete the original library-provided PIO component from your Qsys system.
2. Create the Qsys IP component for the custom ParallelPort component defined in VHDL earlier. This custom ParallelPort will now be available as a new library element in Qsys.
3. Add an instance of your custom ParallelPort to the system and connect it appropriately to the Nios processor. The connections should be identical to the Intel PIO core.
4. Save your Qsys system and go to **Processing->Start Compilation** in the main Quartus window.

5. Program the FPGA with the compiled image.
6. Adapt the software you wrote for controlling the PIO unit from lab 2.0 to match the custom ParallelPort unit you implemented in this lab.

If you are stuck, remember **RTFM**. We stress that *all* the information you need to successfully implement point 2 listed above is fully described in the specified pages of the Quartus Prime Standard Edition Handbook! Don't hesitate to *ask us any questions if something is unclear*. It is essential you understand how all the components in the system are assembled and how they interact.