Lab 8: Interfacing FPGA Spartan-6 with Host Computer via USB

EE-459/500 HDL Based Digital Design with Programmable Logic

Electrical Engineering Department, University at Buffalo

Last update: Cristinel Ababei, 2012

1. Objective

The objective of this lab is to learn one method of implementing communication via USB between the FPGA (Spartan-6 on Atlys board) and the host computer. This method is based on using an excellent open source project called FPGALink [1]. Once this lab is completed you should be able to extend this method and utilize it in any project where you require the computer host to exchange data with the FPGA.

2. Introduction

The Universal Serial Bus (USB) is a specification developed (in the mid-1990s) by Compaq, Intel, Microsoft and NEC, joined later by Hewlett-Packard, Lucent and Philips. The USB was developed as a new means to connect a large number of devices to the PC, and eventually to replace the 'legacy' ports (serial ports, parallel ports, keyboard and mouse connections, joystick ports, midi ports, etc.). USB requires a shielded cable containing 4 wires.

The USB is based on a "tiered star topology" in which there is a single host controller and up to 127 "slave" devices. The host controller is connected to a hub, integrated within the PC, which allows a number of attachment points (referred to as ports). The USB is intended as a bus for devices near to the PC. For applications requiring distance from the PC, another form of connection is needed, such as Ethernet. Note however, that USB is not a true bus: only the root hub sees every signal on the bus. This implies there is no method to monitor upstream communications from a downstream device.

There a lot of online information describing the USB. As a start, you may want to read [2,3].

In this lab we'll use one of the USB ports available on the Atlys board; that is, the so called "Adept USB Port" (see Fig.1), marked as J8 on the board and on the schematic diagram [4]. The USB Controller is a Cypres chip, CY7C68013A-56 USB Microcontroller High-Speed USB Peripheral Controller.

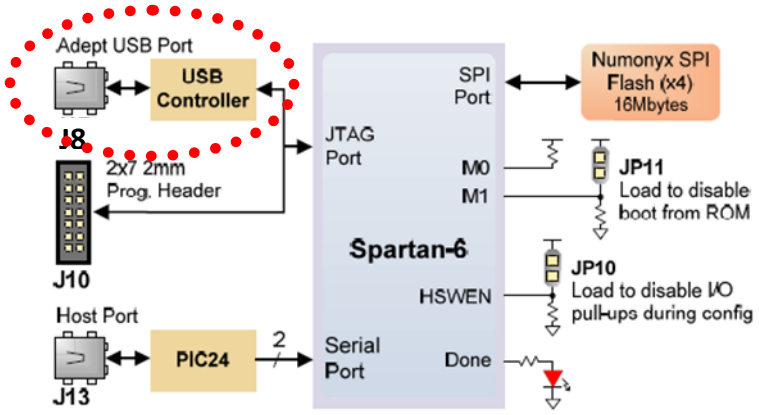

Figure 1

3. FPGALink Library

The FPGALink library was developed by Chris McClelland [1]. It provides an end-to end solution capable of JTAG-programming the FPGA on a variety of USB-based hardware platforms (including Atlys board). It also facilitates communication with the FPGA using a straightforward API on the host side and a standard FIFO interface on the FPGA side.

The FPGALink library is just a C DLL. So, we would normally embed it in our application, for example developed in C/C++ or Python. To get started and help you become familiar with the FPGALink library, the binary distribution archive contains also a utility (called "**flcli**") which provides straightforward command-line access to many of the library functions.

In this lab we will:
1) Use the "flcli" utility to demonstrate the host-FPGA communication using an example that is a slightly changed example that comes with the FPGALink library.
2) Build a simple C++ application to utilize the FPGALink library.

3.1 Working Environment Setup

Notes:
-- Steps 1 and 2 are necessary only if you plan to compile the FPGALink or you are doing this on your personal home computer. Because we'll use the provided downloadable binaries of this library, these steps can be skipped.
-- I have done this lab on Windows (though FPGALink can be used on Linux and Mac too). These steps refer to the Windows.

1) Download and install "Visual C++ Express 2010"
http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express#Visual_Studio_2010_Express_Downloads

2) Download and install "Microsoft Visual C++ 2010 Redistributable Package (x86)"
http://www.microsoft.com/en-us/download/details.aspx?id=5555

3) Download "Build Infrastructure", windows version. This is the environment where we'll work with the FPGALink library binaries.
http://www.makestuff.eu/wordpress/software/build-infrastructure/
On windows, unpack the downloaded archive makestuff-win32-20111211.zip in your own directory.
In my case, I did this directly in C:\. This created C:\makestuff\.

4) Download and install "Console 2". Console is a Windows console window enhancement.
http://sourceforge.net/projects/console
Simply unpack the downloaded archive directly in C:\Program Files\
Then create a shortcut to C:\Program Files\Console2\Console.exe
Launch Console 2 and enter "C:\makestuff\msys\bin\sh.exe --login" in the "Shell" box at Edit->Settings->Console

5) Download the latest FPGALink library binaries (at the time of writing this lab, the latest version is "libfpgalink-20120621.tar.gz (Linux, MacOSX & Win32)"). This is basically the library that we'll use. If your course project will require communication with the host, this will turn out to be very handy.
http://www.makestuff.eu/wordpress/software/fpgalink/
Unpack it in C:\makestuff\libs\

6) Download "LibUSB-Win32". libusb-win32 is a port of the USB library libusb (http://sf.net/projects/libusb/) to 32/64bit Windows (2k, XP, 2003, Vista, Win7, 2008; 98SE/ME for v0.1.12.2). The library allows user space applications to access many USB device on Windows.
http://sourceforge.net/projects/libusb-win32/
Plug in the Atlys board and turn the power on. Then run bin/inf-wizard.exe. Click "Next", select your FPGA board, make a note (in our case, the Atlys board, that is **1443, 0007**) of the vendor and product IDs and click "Next" twice. Choose a location for the driver and click "Save". Click "Install Now".

That's all. We are now ready to use FPGALink library! You should now take the time to read the FPGALink manual:
http://www.swaton.ukfsn.org/docs/fpgalink/vhdl_paper.pdf
FPGALink library comes with two nice examples. Please follow the steps from "README" (C:\makestuff\libs\libfpgalink-20120621\README) to run either of the examples.

3.2 EXAMPLE #1: Communication Host (flcli utility) – FPGA (simple VHDL design)

A) Description

Our application implemented on the FPGA works in this simple example with primarily four registers, referred to as R0, R1, R2, R3. These registers provide the storage space for communicating with the host, and are associated with four different channels of the communication between host and FPGA.

From the host, writes to R0 are simply displayed on the Atlys board's eight LEDs. Reads from R0 return the state of the board's eight slide switches. Writes to R1, R2, and R3 are registered and may be read back. The circuit implemented on the FPGA simply multiplies the R1 with R2 and places the result in R3.

A simplified block diagram of the entire system (host + FPGA) is shown in the Fig.2 below.

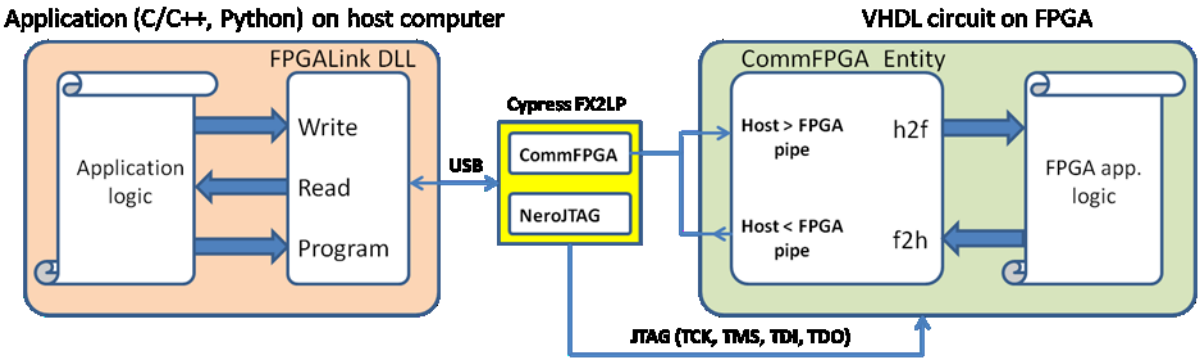

Figure 2 Interfacing the host computer with the FPGA via FPGALink

B) VHDL coding and .xsvf programming file generation

The two VHDL source files (**comm_fpga_fx2.vhdl** and **top_level.vhdl**) together with the .UCF file required to implement the circuit on the FPGA are provided in the downloadable archive of this lab. These files are modified versions of the VHDL example files from the FPGALing library. **top_level.vhdl** is also included in Appendix A at the end of this document.

First, please read these files to understand what they do. Then, create a new ISE WebPack project and add these files to your project. In my case, I called my new project **lab8_usb_fpgalink**. The entire directory of my ISE WebPack project is also included in the downloadable archive of this lab. Synthesize and implement the design.

Generate .xsvf: Method 1

Because we will be programming the FPGA using the **flcli** utility provided as part of the FPGALink library, we need to generate an **.xsvf programming file**. Recall that the FPGA can be programmed using different programming file formats including .bit, .svf, and .xsvf. To generate the .xsvf file follow these steps:

- Inside ISE WebPack, select "Manage Configuration Project (iMPACT)", right-click and choose "Run". The ISE iMPACT window should pop-up after a few seconds.
- Double click on "Boundary Scan" and then File->Initialize Chain. You should get the sc6slx45 "instantiated" in the "Boundary Scan" panel like in this figure:

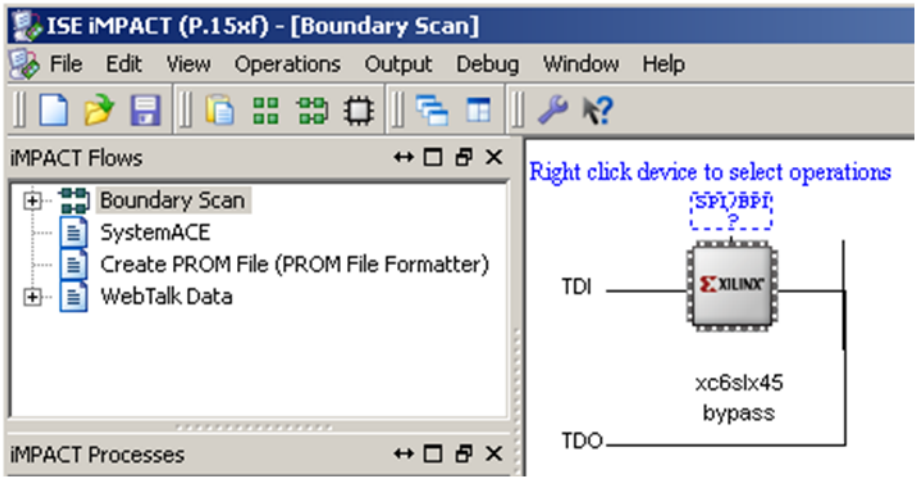

Figure 3

- Right click on the chip and choose Set target device. Assign a configuration file. This is usually a .BIT file such as **top_level.bit** in our case. So, go ahead and select top_level.bit and assign it.
- Select from the menu Option->XSVF File->Create XSVF File… Name it and then click OK to save it in your ISE project directory. In my case I named it **lab8_usb_fpgalink.xsvf**.
- Right click on the chip and choose Program.
- The output will be saved to .XSVF file, **lab8_usb_fpgalink.xsvf**. We'll use this file to program the device.
- Close ISE iMPACT. Close also the ISE WebPack but keep the Atlys board connected and powered-on.

We have now **lab8_usb_fpgalink.xsvf** and so we're ready to program the FPGA and to communicate with it via the **flcli** utility of the FPGALink library binaries distribution.

Generate .xsvf: Method 2

This is optional and meant for the curious. Until now, we've been using Xilinx ISE WebPack tools via the graphical user interface, the actual ISE. However, these tools can be run using Makefiles at the command line too. This alternative approach is especially useful when we want to automate and thereby speed-up the design process: all design steps can be executed via a single makefile. In addition, memory and CPU resource utilization is better. It is left as an exercise for you to read Xilinx and other documentation [5] and to write the simplest makefile required to run the whole process of implementing the design of this lab and to finally generate the .XSVF programming file.

C) Testing and validation of the overall host-FPGA system

Before launching flcli, first create a new folder inside C:\makestuff\libs\libfpgalink-20120621**gen_xsvf** and copy **lab8_usb_fpgalink.xsvf** to it. We'll use the newly created folder, **gen_xsvf**, to store .xsvf programming files of our own projects.

--Connect and power-on the Atlys board if not already.
--Start a terminal by launching Console 2. We'll use the **flcli** utility on the host side.
flcli is a command-line utility, which offers many of the FPGALink library's features. It is useful for testing, etc. Read more about it in the FPGALink manual vhdl_paper.pdf:
http://www.swaton.ukfsn.org/docs/fpgalink/vhdl_paper.pdf

--Use **flcli** utility to program the FPGA. In the Console 2 terminal, do:
> cd libs/libfpgalink-20120621
> ./win32/rel/flcli -v 1443:0007 -i 1443:0007 -s -x gen_xsvf/lab8_usb_fpgalink.xsvf

--Use **flcli** utility to connect to the FPGALink device 1443:0007 (that is the USB controller on the Atlys board):
> ./win32/rel/flcli -v 1443:0007 -c
Which enters the command-line mode, where we can use the **flcli** utility's built-in functions to write and read the registers we have created on the FPGA. For example, try this:
> w0 13
And observe the LEDs on the Atlys board. They should be turned on/off accordingly. Or for example, read the status of the slide switches:
> r0
Write into R1 and R2:
> w1 02;w2 03
> r1
> r2
> r3

If everything went OK, your Console 2 window should look like in Fig.4.
Quit the **flcli** utility:
> q

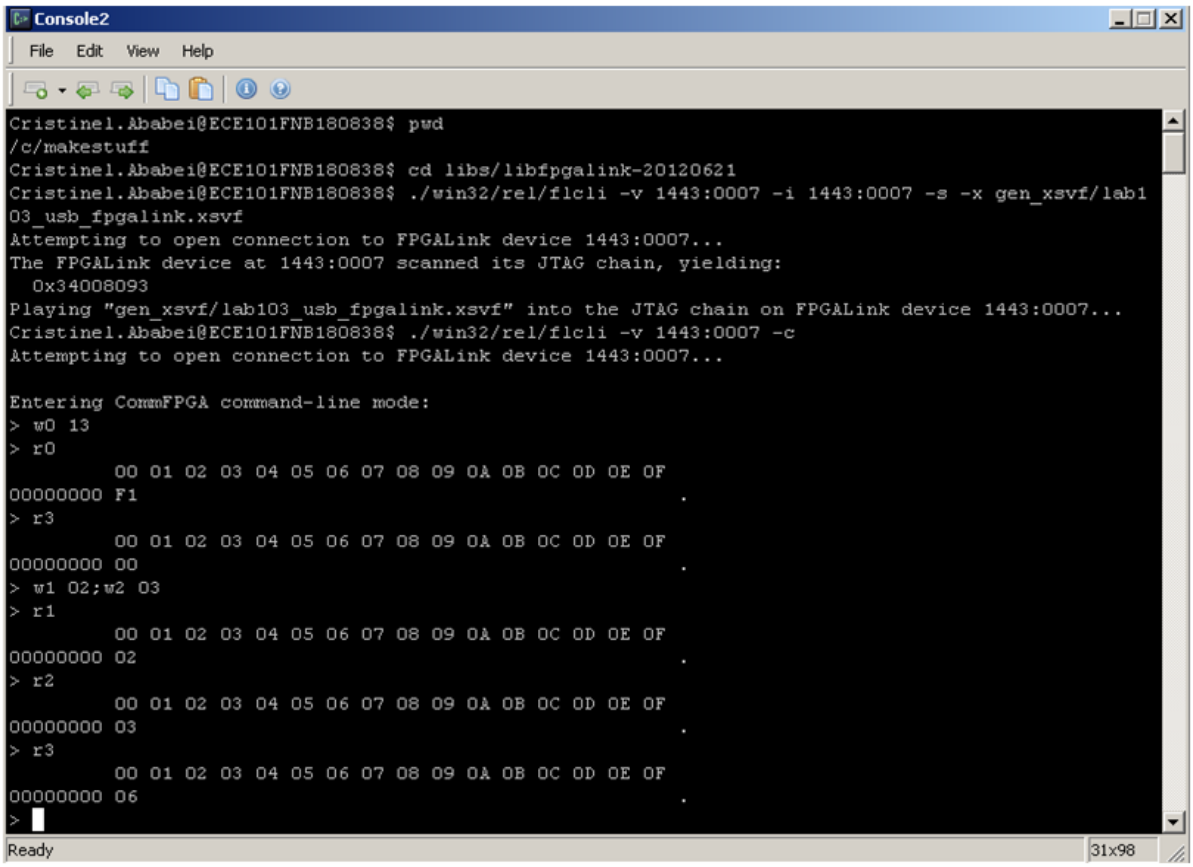

Figure 4 Snap-shot of Console 2 window

4. Lab assignment

Implement a project in which you utilize the FPGALink from your own host-side application written in C/C++ or Python. Your project should open a file file_host2fpga.txt (the file format is with a byte in hex format on each line) and read its content line by line and send it to the FPGA to drive the eight LED. Also, your application should read the eight slide switches and save their status in the same format as above in file_fpga2host.txt. Append a new line to this file each time the switches are changed.

To get started, read first the C example provided as part of the FPGALink binaries distribution. This example is located in: C:\makestuff\libs\libfpgalink-20120621\examples\c

5. Credits and references

[1] Chris McClelland , FPGALink: Easy USB to FPGA Communication.
http://www.makestuff.eu/wordpress/software/fpgalink
[2] USB Home: http://www.usb.org/home
[3] USB Made Simple: http://www.usbmadesimple.co.uk/index.html
[4] Atlys schematic diagram. http://www.digilentinc.com/Data/Products/ATLYS/Atlys_C2_sch.pdf
[5] Xilinx's command line tools user guide.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/devref.pdf
& others: http://www.demandperipherals.com/docs/CmdLineFPGA.pdf

http://outputlogic.com/xcell_using_xilinx_tools/74_xperts_04.pdf

Appendix A: Content of **top_level.vhd** file.

```
--
-- Copyright (C) 2009-2012 Chris McClelland
--
-- This program is free software: you can redistribute it and/or modify
-- it under the terms of the GNU Lesser General Public License as published by
-- the Free Software Foundation, either version 3 of the License, or
-- (at your option) any later version.
--
-- This program is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
-- GNU Lesser General Public License for more details.
--
-- You should have received a copy of the GNU Lesser General Public License
-- along with this program.  If not, see <http://www.gnu.org/licenses/>.
--
-- Additional changes/comments by Cristinel Ababei, 2012
-- Description:
-- From the host, writes to R0 are simply displayed on the Atlys board's
-- eight LEDs. Reads from R0 return the state of the board's eight slide
-- switches. Writes to R1 and R2 are registered and may be read back.
-- The circuit implemented on the FPGA simply multiplies the R1 with R2
-- and places the result in R3. Only reads, from host side, are allowed
-- from from R3; that is an attempt to write into R3 will have no effect.
-- When you input, from host side, data into R1 and R2, data should
-- represent numbers that can be represented on 4 bits only. Because
-- data will have to be input (will be done via the flcli application)
-- in hex, writing for example 07 or A7 into R1 will have the same effect
-- as writing 07 because the four MSB will be discarded inside the
-- VHDL application on FPGA.
--
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top_level is
	port(
		-- FX2 interface -----------------------------------------------------------------------
		fx2Clk_in     : in    std_logic;                    -- 48MHz clock from FX2
		fx2Addr_out   : out   std_logic_vector(1 downto 0); -- select FIFO: "10" for EP6OUT, "11" for EP8IN
		fx2Data_io    : inout std_logic_vector(7 downto 0); -- 8-bit data to/from FX2

		-- When EP6OUT selected:
		fx2Read_out   : out   std_logic;                    -- asserted (active-low) when reading from FX2
		fx2OE_out     : out   std_logic;                    -- asserted (active-low) to tell FX2 to drive bus
		fx2GotData_in : in    std_logic;                    -- asserted (active-high) when FX2 has data for us

		-- When EP8IN selected:
		fx2Write_out  : out   std_logic;                    -- asserted (active-low) when writing to FX2
		fx2GotRoom_in : in    std_logic;                    -- asserted (active-high) when FX2 has room for more data
from us
		fx2PktEnd_out : out   std_logic;                    -- asserted (active-low) when a host read needs to be
committed early

		-- Onboard peripherals -----------------------------------------------------------------
		led_out       : out   std_logic_vector(7 downto 0); -- eight LEDs
		slide_sw_in   : in    std_logic_vector(7 downto 0)  -- eight slide switches
	);
end top_level;

architecture behavioural of top_level is
	-- Channel read/write interface -------------------------------------------------------------
	signal chanAddr  : std_logic_vector(6 downto 0);  -- the selected channel (0-127)

	-- Host >> FPGA pipe:
	signal h2fData   : std_logic_vector(7 downto 0);  -- data lines used when the host writes to a channel
	signal h2fValid  : std_logic;                     -- '1' means "on the next clock rising edge, please accept the data on
h2fData"
	signal h2fReady  : std_logic;                     -- channel logic can drive this low to say "I'm not ready for more data
yet"

	-- Host << FPGA pipe:
	signal f2hData   : std_logic_vector(7 downto 0);  -- data lines used when the host reads from a channel
```

```
        signal f2hValid  : std_logic;                   -- channel logic can drive this low to say "I don't have data ready for
you"
        signal f2hReady  : std_logic;                   -- '1' means "on the next clock rising edge, put your next byte of data
on f2hData"
        -- ------------------------------------------------------------------------------------------

        -- Needed so that the comm_fpga_fx2 module can drive both fx2Read_out and fx2OE_out
        signal fx2Read                : std_logic;

        -- Registers implementing the channels
        signal reg0, reg0_next        : std_logic_vector(7 downto 0)  := x"00";
        signal reg1, reg1_next        : std_logic_vector(7 downto 0)  := x"00";
        signal reg2, reg2_next        : std_logic_vector(7 downto 0)  := x"00";
        signal reg3, reg3_next        : std_logic_vector(7 downto 0)  := x"00";

begin                                                                                   -- BEGIN_SNIPPET(registers)
        -- Infer registers
        process(fx2Clk_in)
        begin
                if ( rising_edge(fx2Clk_in) ) then
                        --checksum <= checksum_next;
                        reg0 <= reg0_next;
                        reg1 <= reg1_next;
                        reg2 <= reg2_next;
                        reg3 <= reg3_next;
                end if;
        end process;

        -- Drive register inputs for each channel when the host is writing
        reg0_next <= h2fData when chanAddr = "0000000" and h2fValid = '1' else reg0;
        reg1_next <= h2fData when chanAddr = "0000001" and h2fValid = '1' else reg1;
        reg2_next <= h2fData when chanAddr = "0000010" and h2fValid = '1' else reg2;
        reg3_next <= std_logic_vector(unsigned(reg1(3 downto 0)) * unsigned(reg2(3 downto 0)));

        -- Select values to return for each channel when the host is reading
        with chanAddr select f2hData <=
                slide_sw_in         when "0000000", -- return status of slide switches when reading R0
                reg1                                 when "0000001",
                reg2                                 when "0000010",
                reg3                                 when "0000011",
                x"00"                          when others;

        -- Assert that there's always data for reading, and always room for writing
        f2hValid <= '1';
        h2fReady <= '1';                                                                --END_SNIPPET(registers)

        -- CommFPGA module
        fx2Read_out <= fx2Read;
        fx2OE_out <= fx2Read;
        fx2Addr_out(1) <= '1';  -- Use EP6OUT/EP8IN, not EP2OUT/EP4IN.
        comm_fpga_fx2 : entity work.comm_fpga_fx2
                port map(
                        -- FX2 interface
                        fx2Clk_in      => fx2Clk_in,
                        fx2FifoSel_out => fx2Addr_out(0),
                        fx2Data_io     => fx2Data_io,
                        fx2Read_out    => fx2Read,
                        fx2GotData_in  => fx2GotData_in,
                        fx2Write_out   => fx2Write_out,
                        fx2GotRoom_in  => fx2GotRoom_in,
                        fx2PktEnd_out  => fx2PktEnd_out,

                        -- Channel read/write interface
                        chanAddr_out   => chanAddr,
                        h2fData_out    => h2fData,
                        h2fValid_out   => h2fValid,
                        h2fReady_in    => h2fReady,
                        f2hData_in     => f2hData,
                        f2hValid_in    => f2hValid,
                        f2hReady_out   => f2hReady
                );

        -- LEDs
        led_out <= reg0;
end behavioural;
```