Practicum 4

[Home] [Back]

Contents

Introduction

[Contents]

In this practicum you are requested to **implement yourself** on FPGA a simple **free-running counter** driven by the external 100 MHz clock available on the Arty development board. The counter is then used as **clock-divider** in order to **turn on/off one of general-purpose LEDs** available on the board with a certain toggle period.

Practicum aims

[Contents]

This practicum should exercise the following concepts:

- review how to implement a free-running counter in Verilog HDL
- run the FPGA implementation flow in *Non Project Mode* from the command line
- automate the FPGA firmware installation in batch mode from the command line
- automate the Quad SPI Flash programming in batch mode from the command line

Navigate to the practicum directory

[Contents]

As a first step, open a **terminal** window and change to the practicum directory:

```
% cd Desktop/lae/fpga/practicum/4_LED_blink
```

List the content of the directory:

```
% ls -l
% ls -la
```

Setting up the work area

[Contents]

Copy from the `.solutions/` directory the main `Makefile` already prepared for you:

```
% cp .solutions/Makefile .
```

Create a new fresh working area:

```
% make area
```

Additionally, recursively copy from the `.solutions/` directory the following design sources and scripts already prepared for you:

```
% cp -r .solutions/bench/    .
% cp -r .solutions/scripts/  .
% cp -r .solutions/xdc/      .
```

> **REMINDER**
>
> In this case we want to perform a **recursive copy**, do not forget to add the `-r` option.
> The above `cp` command works fine under Windows also if you use back-slashes in the path.

RTL coding

[Contents]

Create with your **text-editor** application a new Verilog file named `rtl/LED_blink.v` as follows:

```
% gedit rtl/LED_blink.v &   (for Linux users)

% n++ rtl\LED_blink.v       (for Windows users)
```

The module that we want to implement is a simple **free-running counter** driven by the external on-board 100 MHz clock. We then use the counter as a **clock-divider**, driving a general purpose LED available on the board with a count-slice.

Try to **complete yourself** the following **code skeleton** already prepared for you:

```
`timescale 1ns / 100ps

module (

   input  wire clk,   // assume 100 MHz input clock from on-board oscillator
   output wire LED

   ) ;


   ////////////////////////////
   //   free running counter   //
   ////////////////////////////

   reg [...] count = ... ;   // choose yourself the size of the counter

   always @(posedge clk) begin

      ...                    // increment the counter

   end


   ////////////////////////////
   //   drive the LED output   //
   ////////////////////////////

   // simply turn on/off the LED with a proper count-slice in order to blink the LED with a period approx
   assign LED = ... ;

endmodule
```

In particular, choose yourself the **size of the counter** in order to turn on/off the LED with a **toggle period** of about **one second or more**. Save the source code once done and compile the file to check for syntax errors:

```
% make compile hdl=rtl/LED_blink.v
```

> **QUESTION**
>
> Which is the expected blink frequency of the LED ?
>
> _____

Simulate the design (optional)

[Contents]

Before mapping the RTL code into real FPGA hardware verify the expected functionality of your block with a behavioral simulation:

```
% make sim mode=gui
```

> **IMPORTANT !**

> The default value for the `mode` variable in the `Makefile` has been changed from `gui` to `batch`. If you need to run a flow with the graphical interface use `mode=gui`, otherwise for the remaining of the course the `batch` mode is assumed as default.

Implement the design on target FPGA

[Contents]

Inspect the content of the **Xilinx Design Constraints (XDC)** file already prepared for you:

```
% cat xdc/LED_blink.xdc
```

> **QUESTION**
>
> On which general-purpose LED available on the board is mapped the `LED` module output ?
>
> _____

Verify that all required scripts are in place:

```
% ls -l scripts/build/
```

Map your RTL code on real FPGA hardware running the Xilinx FPGA implementation flow in ***Non Project Mode*** from the command line as follows:

```
% make build
```

Once done, verify that the **bitstream file** has been properly generated:

```
% ls -l work/build/outputs/  | grep .bit
```

Install and debug the firmware

[Contents]

Connect the board to the USB port of your personal computer using a **USB A to micro USB cable**. Verify that the **POWER** status LED turns on. To save time, all **FPGA programming flows** using the Vivado ***Hardware Manager*** have been automated using Tcl scripts.

Verify that all required scripts are in place:

```
% ls -l scripts/install/
```

Assuming that a board is connected to the host computer, **upload the firmware** from the command line using:

```
% make install
```

Verify that the LED blinks as expected.

Program the external Quad SPI Flash memory

[Contents]

As already discussed the firmware loaded into the FPGA is stored into a volatile RAM inside the chip. By default the FPGA configuration is therefore **non-persistent** across power cycles and you have to **re-program the FPGA** whenever you **disconnect the power** from the board.

In order to get the FPGA automatically programmed at power up you have to write the FPGA configuration into a dedicated **external flash memory**. Also this flow has been automated using a Tcl script.

Before running the flow, verify that the **raw binary memory configuration file** has been properly generated:

```
% ls -l work/build/outputs/  | grep .bin
```

Students working with the legacy **Arty** board can program the external **128 MB Quad SPI Flash memory** as follows:

```
% make install_flash board=Arty
```

For students working with the new **Arty A7** board revision instead:

```
% make install_flash board=ArtyA7
```

Disconnect and reconnect the USB cable from your computer and verify that the firmware is properly loaded from the external Quad SPI Flash memory at FPGA startup.

Exercise

[Contents]

Modify the **constraints file** `xdc/LED_blink.xdc` in order to map the `LED` output on the **PMOD header JA1 pin** as follows:

```
#set_property -dict {PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports LED]
set_property -dict { PACKAGE_PIN G13  IOSTANDARD LVCMOS33 } [get_ports LED]
```

Save the file once done and re-run the flows from scratch up to FPGA programming with:

```
% make clean
% make build install
```

Probe the JA1 pin at the **oscilloscope** and verify that the **frequency** of the displayed clock waveform is the expected one as implemented in your RTL code.