ECE 5727

Lab 1: Binary Counter

Lab Point Value: 20
Post-Lab Point Value: 35

Objectives

In this lab you will:

1. See how to send hardware interrupts to the processor
2. Learn about GPIO - buttons and LEDs

Lab Overview

This lab will consist of two parts. Part one is doing Zynq Book tutorial 2 to create a binary counter using the 4 GPIO LEDs on the board. Pushing one of the four buttons will generate an interrupt to be added to the hardware interrupt queue for software processing. A programmable timer running on the FPGA will also generate an interrupt, incrementing the counter by 1 each time it does so.

Why do we need software to maintain and update a counter? Do we really want to interrupt our main program every time a button is pressed? The second part of our lab will be re-implementing this lab, but this time doing so solely on the FPGA.

Instructions

Part One

(*5 points*) Work your way through Exercise 2A-D: Next Steps in Zynq SoC Design.

Part Two

We will now replicate the Zynq Book Exercise 2 solely on the FPGA. Begin by creating new Vivado project, importing the provided files.

- Select to create an RTL project, making sure to uncheck 'Do not specify design sources at this time'
- Add button_counter_top.v, button_counter_tb.v, and clk_gen.xci as sources. Under the column titled 'HDL Source For' change button_counter_tb.v from 'Synthesis & Simulation' to 'Simulation Only'
- Add button_counter.xdc as a constraints file

- Be sure to select the Zybo-Z7-10 as the board and click through to finish.

For now, don't worry about what *clk_gen.xci* and *button_counter.xdc* are for. We will learn about these in later weeks.

button_counter_tb.v is the testbench file that provides basic stimulus for the testing the main Verilog code. You can change it to help you in the lab, though you likely shouldn't need to. You will need to update it for the post-lab.

button_counter_top.v is the main design shell for you to work with. You should not need to add any additional wires/regs to the module. All work you do should merely be to implement logic within the appropriate **always** block.

1. (*10 points*) Implement the button increment feature
   (a) Begin by adding code to detect when a button is pressed and increment the counter. Run the code through synthesis.
   (b) Run a simulation using the provided test bench to confirm functionality. *Submit a screenshot of the simulation waveform - 5 pts*
   (c) Update your code to increment the counter based on button number, like in Part One. Button 0 increments by 1, Button 1 increments by 2, etc. Run synthesis, then simulation again. *Submit a screenshot of the simulation waveform - 5 pts*
   (d) Generate a bitstream and load it into the FPGA to test your work. Does your design work perfectly? Or does the counter sometimes act as if the button is pressed multiple times?
   (e) Switch the *assign btns* logic to use the values from db_btns instead of BUTTONS.
   (f) Re-generate the bitstream and test again. Does the design work better now? Change the value of DB_TIME (by at least an order of magnitude), then re-generate to see how that affects the performance of your design. Continue playing with this value until you find the performance you like.
2. (*5 points*) Implement the timer feature
   (a) Add code to run a timer. Run through synthesis.
   (b) Run simulation. I highly recommend you adjust the localparam TICK_PER_MS and/or TIMER_MAX to much smaller values so the simulation does not need to run as long. *Submit a screenshot of the simulation waveform - 5 pts*
3. (*5 points*) Submit your team's final code, along with the simulation screenshots (there should be 3).

Post Lab

Answer the following questions, and submit along with any supporting materials (screenshots, testbench code, etc).

1. (*15 points*) Take a look at the debouncer logic driving db_btns. Modify the testbench to examine this logic, and submit supporting screenshots of the waveform to support your answers. When running the simulation, also make sure to reduce the value of DB_TIME to a very small value (like 5-10), otherwise your simulation will need a lot of time to finish, making for difficult screenshots.

    - What is the logic doing?
    - How does this help prevent the counter from occasionally over-incrementing on button presses?

2. (*15 points*) Explore writing reset logic in various ways. As you go through the below steps, compare how the different styles affect the resulting design. Write up a summary of your observations of how different styles impact the final design.

    (a) All your logic should already be in the **always** block with low-priority synchronous reset. Look at the RTL and post-synthesis schematics.

    (b) Copy your logic into the **always** block with the synchronous reset with priority over clocked logic. (Comment out the other **always** blocks). Look at the RTL and post-synthesis schematics. How does this differ from what you originally had? (Hint: take a look at the btn_last flops)

    (c) Now copy your logic in the asynchronous reset block, commenting out the synchronous blocks. Look at the RTL and post-synthesis schematics. How does changing to an asynchronous reset affect the design compared to the two synchronous versions? (Hint: take a look at the counter/timer flops)

3. (*5 points*) We have successfully replicated the binary counter from the ZynqBook Exercise 2 in all ways but one. In Exercise 2, the software has the ability to dynamically set how frequently the timer generates an interrupt, and therefore the timing for automatically incrementing the counter without a button press. As it is currently written, our timer value is specified using a localparam, which means that it must be determined prior to loading the design onto the FPGA and cannot be updated at runtime. How might we change the implementation of the timer so that it can be dynamically configured? *There isn't necessarily a correct answer here - I'm curious to see what you come up with; you can assume you have controlling software if you'd like*