1 Prac 4 - FPGA Introduction

1.1 Introduction

For this practical, it is important to complete the tutorial in order to get acquainted with the tools required to complete the practical. It is recommended that you work in groups of 3, and that all three members work through the tutorial individually.

It is also imperative that you read through the wiki. Operation of Vivado can be intimidating at first use. The Wiki has been carefully written to include all you need for the practicals.

1.2 Tutorial

The tutorial is not for marks but it is suggested you complete it, as it will give you the basics of Vivado, and allow you to upload a simple program to the FPGA. In the tutorial (as on the wiki) we use the Digilent Nexys A7 as an example board, but the process should be the same for the Nexys 4 DDR and Nexys 4. The only thing that will change between the three is the board you select when creating the project, and the constraints file you use. Note the constraints file does determine naming of some of the I/O, so double check that.

Most of the details on how to complete these steps are on the wiki under Xilinx Vivado.

1. Install Vivado
2. Install boards
   The only boards you might work with in this course are the Nexys 4, Nexys 4 DDR, and the Nexys A7. So you only need to worry about installing those.
3. Download and save the constraint files. That, or copy the simple example given on the wiki if you're using the A7.
4. Create a new project.
   - You can call it "Tutorial".
   - Set it as an RTL project and select "Do not add sources at this time". For the prac later, you can add the given source files here.
   - Select your board appropriately.
5. Add constraint source
   - Right click on constraints, select "Add sources". In the dialog box, make sure "Add or create constraints" is selected. Hit next.
   - Select "Create file" if you are copy pasting the constraints in, or "add files" if you have the constraints file downloaded locally. Press "Finish".

- If you were intending on copy-pasting constraints in, do so now. Ensure your constraints are correct for the board, and have the clock, two switches and two LEDs enabled.
- Right click on the constraints file, and select "Set as Target Constraint File"

6. Add the Verilog source
   - Right click on "Design Sources, select "Add sources". In the dialog box, make sure "add or create constraints" is selected. Hit next.
   - Select "Create file". Call it the same name as your project (good practice for the top level module). Press "Finish".
   - A dialog will open, with ports. You can just press "okay", as we'll define ports in the Verilog code.
   - Right click on the Verilog file, and select "Set as Top" (if it is not already set as top - indicated by being in bold).
   - In it, paste code to, each clock cycle, write switch[0] to LED[0] and the inverse of switch[1] to LED[1]. Example code can be found on the wiki.
7. Select "Run Synthesis"
8. Select "Run Implementation"
9. Select "Generate Bitstream"
10. Upload your bitstream to your target board (speak to a tutor to get a board)
    (a) Plug in the board
    (b) Select "Open Hardware Manager"
    (c) Select "Open Target" and then "Auto Connect"
    (d) Vivado should find the board. Select "Program Device" and select the board you've plugged in (The A7 shows as "xc7a100t_0". Press the "Program" button.
11. Hooray! You've created your first FPGA circuit. Toggle the switches to see if it operates as expected. Now let's move on to the fun stuff.

1.3 Practical

1.3.1 Introduction

In this practical, you will create a digital clock on an FPGA. There is no report for the practical, but you will need to submit screenshots of your testbenches and demonstrate your implementation to a tutor.

Source files are available on the EEE4120F OCW GitHub: https://github.com/UCT-EE-OCW/EEE4120F-Pracs.

1.3.2 Given Modules

1. TLM
   The top level module, called "Clock.v" in the source files on GitHub, contains the primary logic for your wall clock and allows you to implement I/O and other modules

2. Delay_Reset
   It's also useful as many components require a set up time. So by using a delayed reset signal, we can cater for reset times of peripherals.

3. Seven-Segment Driver
   This module takes 4 BCD values and displays them on the seven segment display.

4. Decoder
   Used by the Seven-Segment Driver to decode decimal to the appropriate cathode pins.

5. Debounce
   A debounce module you'll need to implement in order to debounce button presses.

6. PWM
   A module you'll need to implement in order to give the seven segment displays changing brightness. This can be tricky, it's suggested you leave it for last.

1.3.3 Requirements

The following outcomes are required to pass the demonstration:

1. Implement a simple state-machine to display the real time (hours and minutes) on the 7-segments display. You can start your clock at 00:00 upon reset. Make your clock faster in order to test that the time overflows correctly. Use a 24-hour time format.

   The easiest way to do this is with deeply nested if statements. Run a counter that overflows every second and increment the seconds counter on every overflow. Every time this seconds counter equals 59 (i.e. it will overflow on this clock-cycle), increment a minutes units counter. Every time this minutes units counter is about to overflow, increment a minutes tens counter, etc.

   Only use non-blocking assignments (<=). Blocking assignments (=) inside clocked structures are much more difficult to debug. Remember that the entire always block is evaluated at once: the statements are not evaluated sequentially.

2. Display the seconds on the LEDs, in binary format. This is done with a simple assignment outside the always block.

3. Use one of the buttons, properly debounced, to set the minutes. It must increment time by one minute every time it is pressed. Make sure that your time overflows correctly. You do not need to increment the hours when changing the minutes.

   It is recommended to write a Debounce module for this. On every clock cycle of the system clock (the fast one), check the state of the button. If it is not the same as the

current module output, change the output and start a dead time counter. While this counter is counting, do not change the output of the module, no matter what the input is doing. Use a deadtime of between 20 ms and 40 ms. To prevent unstable states, register the button before use.

In the clock state machine, you can use a register to store the button's 'previous' state. If the current state is high, and the previous state is low, the button signal went through a rising-edge. Do not use always @(posedge Button) – keep everything in the same clock domain.

4. Use another one of the buttons, properly debounced, to set the hours. It must increment time by one hour every time it is pressed. Make sure that your time overflows correctly.

5. Use the slide-switches to represent a binary "brightness" word. Make use of pulse-width modulation (PWM) to dim the brightness of the LED display.

   Ensure that the phasing between the driver signals and the PWM signals is correct. The easiest way to do this is to select a PWM frequency such that the PWM signal goes through exactly one period between driver signal state changes. You can implement this within the SS Driver module.

1.4 Mark Allocations

Table I: Prac 4 mark allocation

		Marks
Testbench		12
	Minute seconds reaching 60 and increasing minutes	
	Minutes reaching 59 and increasing hours	
	Time reaching 23:59 and wrapping back to 00:00	
	(3 each +3 for neatness)	
Demo		
	Time flows correctly (minutes overflow to an increase in hours, and 23:59 flows to 00:00) [6, subtracting 2 for each missed objective]	6
	Time can be scaled through a variable (i.e. the count to increase seconds isn't fixed)	2
	Minute button increases minutes and doesn't increase hours	2
	Hour button increases hours	1
	Debounce module implemented	2
	PWM module	5
	TOTAL	30

For the 5 marks on PWM: 1 mark for attempted, 2 marks for reading switches and adjusting, 3 marks for "flashing" implementation, 4 marks for some mix between flashing and decent PWM, 5 marks for correctly implemented.