ES 4 Lab 6: Flashing Displays

Prelab due 24 hours before your lab session, week of Oct 20
Lab report due at your lab time, week of Nov 3

1 Introduction

In this lab, you'll combine a sequential circuit with some more clever combinational logic to display a 2-digit number in base 10 on your seven-segment display. While the final product isn't exactly mind-blowing, it pulls together several important techniques and may provide a springboard for some key components of your final project.

After successfully completing this lab, you should be able to:

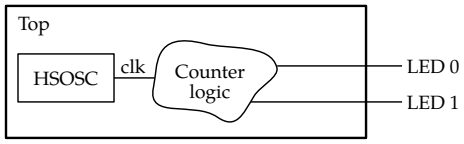
- Use the FPGA's internal oscillator to drive sequential logic
- Divide the clock to toggle signals with varying frequencies
- Divide binary numbers on the FPGA by multiplying and shifting

2 Prelab

You can complete this first section of the prelab either on VHDLweb or using your FPGA. We encourage you to use the FPGA, since it'll set you up to continue straight into the lab tasks.

2.1 Flashing LEDs

First we'll build a VHDL module which alternately flashes two LEDs.



P1: Create your top-level VHDL entity and set up the clock signal. The `HSOSC` module is a special VHDL module which represents the high-speed oscillator within the iCE40 FPGA. By default, it produces a 48 MHz clock signal on the `CLKHF` output.

You can instantiate the following VHDL component, and connect your clock signal (`clk`) to the `CLKHF` output. The `CLKHFPU` and `CLKHFEN` signals should be driven with a constant value `'1'`.

```
component HSOSC is
generic (
    CLKHF_DIV : String := "0b00"); -- Divide 48MHz clock by 2^N (0-3)
port(
    CLKHFPU : in std_logic := 'X'; -- Set to 1 to power up
    CLKHFEN : in std_logic := 'X'; -- Set to 1 to enable output
    CLKHF : out std_logic := 'X'); -- Clock output
end component;
```

`HSOSC` is defined in a library that's included with Radiant, so you just need this component declaration to reference it.

This may be the first time you've seen a `generic`. While a `port` represents an input or output signal that can change dynamically in the final design, a `generic` is a *synthesis-time* parameter that configures how the component behaves.[1] In this case, you can configure the frequency of the clock by changing the value of the `CLKHF_DIV` generic, but this configuration happens when you *synthesize* your design and cannot be changed dynamically after the design is flashed. If you don't specify anything, it'll just use the default.

An example instantiation is shown below:

```
osc : HSOSC generic map ( CLKHF_DIV => "0b00")
            port map (CLKHFPU => '1',
                      CLKHFEN => '1',
                      CLKHF => clk);
```

P2: Build a 26-bit binary counter. On the rising edge of each clock, the counter value should increment by one. When it reaches the maximum value, it will just roll over to zero.

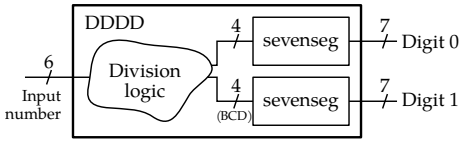
P3: Finally, take the top bit of the counter (`counter(25)`) and use this to drive the two LEDs. One LED should be on when the counter bit is high; the other should be on when it is low. Because the counter rolls over (overflows back to zero), this top bit will slowly alternate between 0 and 1.

P4: If you're using the UPduino, you should see the LEDs togging on and off. How long is one on-off cycle of an LED? *You should calculate this based on what you know about the oscillator and the counter, and check it against what you see.*

We're going to use this trick to drive the two digits of the seven-segment display. Because the two digits share the cathode wires, we can't turn both digits on and control them independently. However, we can display a value on one digit, while keeping the other digit off (by keeping the anode low). Then we swap, and display the value on the second digit while keeping the first one off. If we do this quickly enough, it will appear to your eyes that both digits are on, even though they're really flashing back and forth.

2.2 Dual-decimal-digit-driver (the DDDD)

P5: Write VHDL for an entity named `dddd` that takes a 6-bit unsigned integer and produces the output wires for two seven-segment displays.



You should reuse your seven-segment display module from the previous lab, and just instantiate it as a component in the module you're building.

The crux of this task is doing the math to get a binary number representing each digit[2]. Calculating the one's digit only requires the `mod` operator, which our tools can map onto the FPGA relatively easily.

[1] Kind of like the difference between a #define versus a variable in C.

[2] This is sometimes called "binary coded decimal" (BCD), because we're representing a decimal number by coding each digit in binary.

However, division by factors other than 2 is quite hard[3]. Instead of trying to divide by 10, we'll multiply by 52 and divide by 512 (and drop any fraction). This is equivalent to dividing by 10.16, which is pretty close to 10. You can do this with VHDL code like the following:

```
-- Do the math to split up the digits.  Input `count` is 6 bit unsigned
lowBCD <= count mod 4d"10"; -- Low digit result is 4 bit unsigned
-- Multiply by 52. Intermediate term is 13 bit unsigned
tensplace <= count * 7d"52";
-- Divide by 512 (2^9). High digit result is 4 bit unsigned
highBCD <= tensplace(12 downto 9);
```

P6: Describe what kind of numeric input you want to use to drive your shiny new double-digit display. You have several options — we want to give you opportunity to experiment and build something cool, while also providing a simpler alternative if you're swamped this week.

- Use 6 bits from the DIP switch as a 6-bit number
- Reuse your ALU from Lab 3, and make the result 5 bits so that it doesn't overflow.
- Use one or more pushbuttons to control a simple counter. Hints on this are on the course website.
- Use a rotary encoder (i.e., a knob) to drive a counter. Instructions for this are on the course website.

Note: your lab kit has everything you need to complete the problems below. You're welcome to get a head start on it before you come to lab!

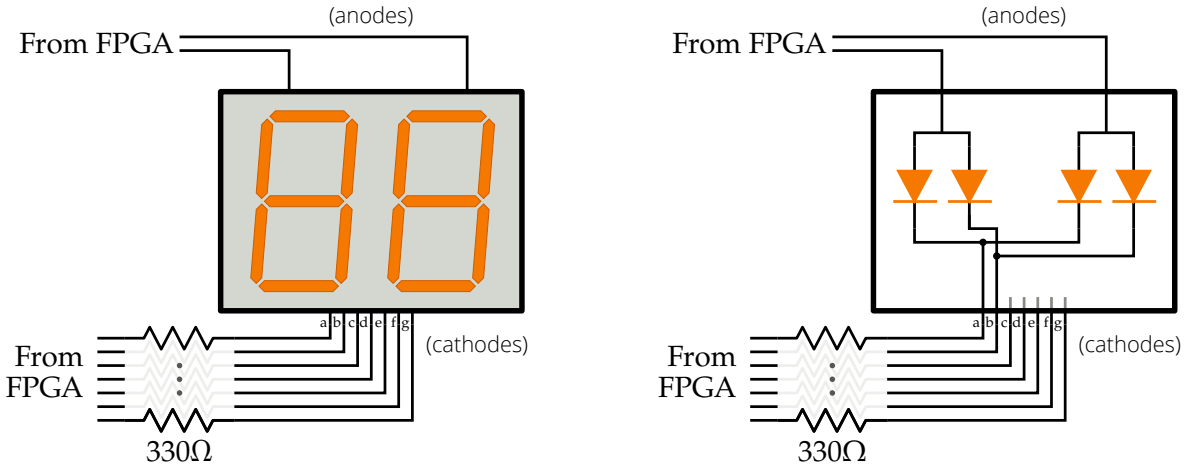
[3]So hard, in fact, that Radiant will refuse to do it, saying "`0 definitions of operator "/" match here`". On the other hand, dividing by 2 is easy: just drop the lowest bit and consider the rest of the bits to be one place lower.

3 In lab

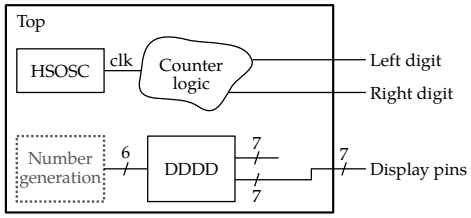
L1: If you haven't already, run your LED-blinking design on the UPduino. Increase the flashing frequency of the LEDs by selecting lower bits of the counter. How fast do you have to go before they don't appear to be blinking anymore?

3.1 Two-digit display

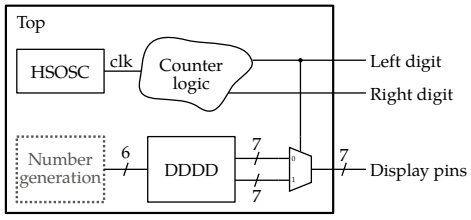
L2: Wire up the seven-segment display following the diagram below. The diagram on the right shows how the LEDs are wired internally (only two LEDs on each digit are shown for clarity).



L3: Update your LED blinking module to instantiate the DDDD. It's up to you how you'll drive the input to the DDDD. Whatever you've chosen, we suggest that for initial testing you just use a group of input pins that you control with the DIP switch or Digital Discovery.



L4: Finally, write VHDL to toggle the number being displayed based on which LED output is on.



If it works, you should have a fully-functional 2-digit display — controlling 14 LEDs with only 9 outputs.

> Magic!

> **L5**: **Optional**: We've assumed a 6-bit input, giving numbers from 0 to 63. But with two digits, we could potentially display numbers up to 99. Update your DDDD module to handle a 7-bit input, assuming that it only needs to work up to 99.

4 What to turn in

Your lab report should contain:

- Standard "front matter" (see the lab reports handout).
- A photograph of your completed circuit.
- A description of your design. At a minimum, this includes a block diagram and an explanation of the inputs you used for the display. You may reference the VHDL code in this section, but the VHDL code should not stand on its own. Include any other key information needed to understand or replicate your design.
- Your complete VHDL code. Submit your files at the end of your lab report. Please include the code as text, and not as a screenshot. It's possible to copy text out of a PDF in case we want to test or debug your code; a screenshot makes that impossible.
- Your lab journal, including intermediate tests you performed (including ones that failed), steps you took to debug your design, and problems you encountered.
- Your plan(s) for testing your design, and any test results you recorded.
- Answers to the following questions:
  - What was the most valuable thing you learned, and why?
  - What skills or concepts are you still struggling with? What will you do to learn or practice these?
  - How long did it take you to complete the lab?