ES 4 Lab 6: Animations using read-only memory

1 Introduction

In this lab, you'll combine sequential-logic building blocks to create a blinking animation on either your 7-segment display or discrete LEDs. A read-only-memory (ROM) will store a sequence of frames for your animation, and a counter (or at least some bits of a counter) will be used as the address for the ROM. As the counter counts, it selects different frames from the ROM, causing your animation to flash on the LEDs.

Both counters counters and memories (and counters that index into memories) are important building blocks many digital designs, including the ARM microprocessor. This lab will also give you practice using hierarchical design with multiple VHDL entities, which will be crucial for larger designs.

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

- Write VHDL code which will *infer* a read-only memory block in your FPGA.
- Use a counter to iterate through memory addresses and read the corresponding values.

2 Prelab

There is no prelab for this lab since we are working remotely. Read through the whole lab handout to get the big picture, and then dive in!

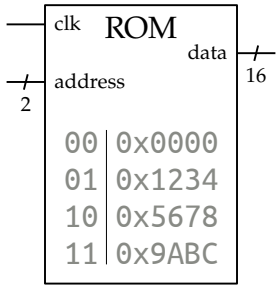
3 In lab

L1: Connect up some LEDs to your UPduino. One simple option is to keep your hardware layout from lab 3, and just use one of the 7-segment display sections.

L2: Decide what kind of animation you want to display. You could show a flashing or "chasing" pattern, display a sequence of letters or numbers, or something else. You should have at least 8 frames, but the FPGA has 1 megabit of memory ($\tilde{1}$12k 8-bit frames) if you want to get creative.

3.1 Frame memory

If you only need a ROM, the basic behavior of the memory embedded in the FPGA is relatively simple. The memory takes an address and clock as inputs, and provides the data as an output. This is shown below for a hypothetical ROM with 2 bits of address and 16-bit data words:



The address is read on the rising edge of the clock and the data appears on the output port after a brief propagation delay. In this example, the 2 address bits specify four possible addresses, each of which refers to a 16-bit word, for a total of 64 bits of storage. Both the address width and the data width can be customized.

To use one of the built-in memory blocks in the FPGA (known as embedded block RAMs, or EBRs), simply write VHDL code which mimics this behavior: on the rising edge of the clock, use the address to select a corresponding row of data.

```
process(clk) is
begin
  if rising_edge(clk) then
    case addr is
      when "00" => data <= 16x"0000"; -- Assumes 2-bit address and 16-bit data
      when "01" => data <= 16x"1234"; -- You can make these any size you want
      when "10" => data <= 16x"5678";
      when "11" => data <= 16x"9ABC";
      when others => data <= 16x"0000"; -- Don't forget the "others" case!
    end case;
  end if;
end process;
```

The synthesis tool will automatically *infer* that you're creating a memory, and use a memory block instead of trying to just build it with regular logic. There are two caveats:

- You have to use exactly this design pattern. If you code something else that doesn't match the behavior of a memory, then the synthesis tool will build that, and not a memory. Modern synthesis tools are pretty clever, but they can't read your mind.
- If your data is really small, the synthesis tool will build your hardware using regular logic elements instead of instantiating an EBR. In my experiments, the synthesis tool started using an EBR above 16 frames of 7 bits each.[1]

> **L3**: Write the VHDL necessary to encode your animation as a ROM. If you have a complex animation, consider writing a script or program to generate the VHDL for you. An example Python snippet to this for a string of digits is shown below, but you could easily achieve the same thing in Matlab, C++, or even Excel.

```
import numpy as np

romstring = "31415926535897932384626433832795028841971693993751058209749445 92"
addrbits = int(np.ceil(np.log2(len(romstring)))) # How many bits of address?

for i in range(len(romstring)):
  # Depending on what your string contains, you may need more sophisticated
  # conversion of the character/number to a bit pattern.
  print('    when "{}" => data <= 7d"{}";'.format(
                  np.binary_repr(i, addrbits), romstring[i]))
```

3.2 Counter

> **L4**: Now you need something to generate addresses at human-visible speed (between 2 and 50 Hz). One easy way to do this is to build a large counter (say 26 bits) driven by the HSOSC and just

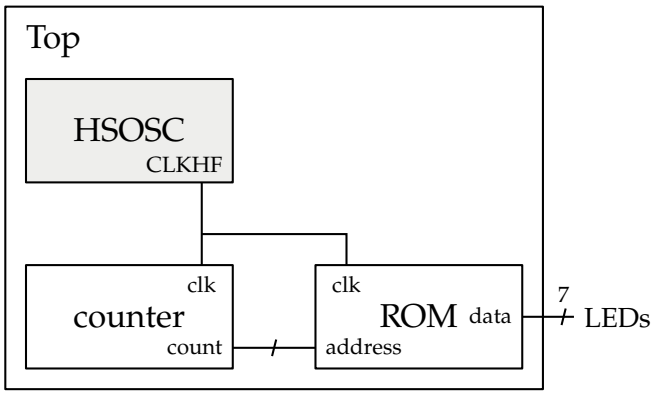
[1] It's fine for the purposes of this lab if you have less data than that and don't actually use the EBR as a result, as long as your animation works.

take the top few bits as the address.

Note that if you have a number of frames that is not a power of 2, you'll either have some dead time when the animation cycles around, or you'll need to make the counter reset to zero when it gets to the end of the animation, rather than simply rolling over.

3.3 Implementation

L5: Build a top-level module which incorporates your memory and your counter as shown below.



L6: Compile your design. After running place & route, check whether the tool used an EBR for your ROM. You can do this by opening the "Reports" tab in Radiant (View → Reports) and looking at the EBR count under "Resource Usage". If your animation data is small, it's possible that no EBRs are used, but normally this count should be 1 or more.

L7: Finally, run it on your FPGA. We'd love to see it working, so take a video to share!

Getting the pin assignments right could easily become the most frustrating part of this lab if you're not methodical. Think about what kinds of patterns you could display to make it easier to sort out which is which.

3.4 Possible extensions

There are lots of fun and simple ways you could extend this project, if you're so inclined.

- Use both digits on the 7-segment display, by reusing your display-flashing logic from lab 4 and doubling the data width stored in the memory.
- Use an LFSR instead of a binary counter so that the animation displays pseudo-random frames.
- If you run your animation really fast (say 1 kHz) and flash sements on and off, you can achieve varying brightness. An even better way would be to store brightness values (say 8 bits) in the memory and build a PWM controller to modulate each LED. An example PWM controller problem is on VHDLweb.

4 What to turn in

Your lab report should contain:

- Standard "front matter" (see the lab reports handout).
- A photograph of your completed circuit (or even better, send your TA a video or GIF of your animation!)
- A description of problems you encountered, your debugging process for identifying them, and your solutions.
- 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? This will help calibrate the workload for future iterations of the course.
- Your VHDL code, at the end of the report. If you used a script to generate VHDL, include your script, and not your bazillion lines of generated VHDL.