ES 4 Lab 7: Using RAM

1 Introduction

In this lab, you'll use the iCE40's built-in RAM (read/write memory) to create the visual equivalent of an audio looper. The finished design will have a set of LEDs with corresponding buttons. When a user holds down the "record" button, the FPGA continuously records the state of the input buttons. When the "record" button is released, the FPGA goes into playback mode and repeatedly plays the button sequence on a set of LEDs.

Both counters counters and memories (and counters that index into memories) are important building blocks many digital designs. 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:

- Use VHDL code which will *infer* a dual-ported RAM memory block in your FPGA.
- Use a counter to iterate through memory addresses and read the corresponding values.
- Write non-trivial logic to control the behavior of multiple counters.

2 Prelab

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

3 In lab

> **L1**: Decide how many channels you want, and hook up an appropriate number of buttons and LEDs. For N channels, you'll need N leds and N+1 buttons (one button to control recording, plus one for each channel). It is perfectly acceptable to just use one channel, but more is fun!

> **L2**: Decide how many times per second you want to sample the buttons (we'll call this the "sample rate"), and how long you want to be able to record for. How many bits of storage do you need? The FPGA has 1 megabit of memory, but you shouldn't need anywhere near this much.

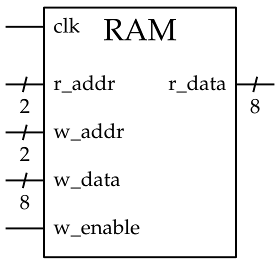
3.1 Instantiating memory

The RAM design we'll use here has two ports, meaning that it's possible to read data out from one address while writing into another.[1]

A hypothetical RAM with 8-bit words and 2-bit addresses is shown below.

On the rising edge of the clock, the RAM saves the input data (`w_data`) to the location specified by write address (`w_addr`) if the write enable signal (`w_enable`) is high. On the same rising edge, it also reads the word selected by the read address (`r_addr`) and returns the data as an output (`r_data`).

[1]We're going to use it as if it were a single-ported memory, but seeing how a dual-ported memory works may be useful in your final project.



In this example, the 2 address bits specify four possible addresses, each of which refers to an 8-bit word, for a total of 32 bits of storage. Both the address width and the data width can be customized.

To use the built-in memory blocks in the FPGA (known as embedded block RAMs, or EBRs), simply write VHDL code which mimics this behavior. An example RAM module is posted on the course website which you can instantiate directly and customize using VHDL generics.

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 (a few dozen bits), the synthesis tool will build your hardware using regular logic elements instead of instantiating an EBR.

L3: Download the dual-ported RAM module from the course website (`ramdp.vhd`).

Create a top module, and instantiate the RAM as a component inside the top module.

3.2 Counter

L4: Now you need something to generate addresses at your desired sample rate.

1. Instantiate the HSOSC as a component just like in lab 6.
2. Build a counter driven by the 48 MHz HSOSC clock that rolls over once per looper sample. You can do this by just making the counter a certain number of bits and letting it roll over naturally, or you can make it reset when it reaches a particular value. The first is simpler but the rate will be a power of 2; the second gives you precise control of the rate at the expense of a little more logic.
3. Build a second counter which increments every time the first counter rolls over. You may be tempted to do this by using the first counter as a clock:

```vhdl
-- Don't do this!!
process(fastcounter(12)) is
begin
  if rising_edge(fastcounter(12)) then
    sample_counter <= sample_counter + 1;
  end if;
end process;
```

   This may work, but it's better to keep all of the logic in the FPGA on the same clock. To accomplish this, you can keep your `process` dependent on `clk` but make the update conditional on the first counter:

```
process(clk) is
begin
  if rising_edge(clk) then
    if fastcounter = 13d"0" then -- Counter just rolled over
      sample_counter <= sample_counter + 1;
    end if;
  end if;
end process;
```

This is a design pattern you will use many times in larger designs, particularly your final project.

3.3 Wiring everything up

L5: Get the looper working in its most basic form by writing VHDL code to do the following:

1. Connect the record button input to the read/write signal on the memory.
2. Connect the channel button input(s) to the RAM input (write value).
3. Connect the LED output(s) to the RAM output (read value) when in play mode, and to the buttons when in record mode (this way you can see what you're recording!)
4. Connect the memory read and write addresses to the sample counter you just built.

With this working, you should be able to record button pushes and see them play back a few seconds later! Note that you'll have to wait your entire recording length until the counter rolls over to the beginning.

L6: The looper ought to resume playback at the beginning as soon as recording is finished. To do this, you'll need to be able to tell when the button was pushed or released.

1. Create a new signal that holds the value of the control button on the *last* clock cycle.
2. Use the current value of the control button together with its previous value to determine if it was just pressed or released.
3. Write appropriate conditional statements to reset the sample counter when recording starts and stops.

Now you should be able to record a pattern and see it play back immediately after you release the record button. However, you'll still have to wait for the whole recording length before the pattern repeats.

L7: Finally, the looper should loop over the part that was recorded and not the entire available memory. To do this you'll need a register to store the length of the recording.

1. When the control button is released, save the current value of the sample counter. This is the length of the recording.
2. When the system is in playback and the sample counter reaches the end of the recording, it should start back at the beginning. (If the system is recording, you'll simply ignore the previously recorded length.)

> If it works, you should be able to record a pattern of any length and watch it repeat endlessly! Great work!

4 What to turn in

For this lab, you do not need to write up and turn in a full report; we want you to focus on the final project.

Instead, please turn in an abbreviated report with the following:

- Standard “front matter” (see the lab reports handout).
- A video or GIF of your completed looper would be super cool, but not necessary
- Lab journal pages (you *are* still taking notes, right?)
- Your VHDL code, at the end of the report. You don’t need to include the supplied RAM module unless you’ve modified it.