Computer Science 141
Lab #7: Edge-based Clocking & RAM

For Lab: November 5 / November 6

This lab will get you started creating edge-based clocking designs. You will design a 2Kx32bit RAM block using RAM modules already implemented in the FPGA. We will also add support for a full-memory reset.

This lab will be more challenging than previous labs. You should try the lab out before your lab section, but we expect that you will need some in-person assistance for the new topics.

1 New Concepts: Edge-based Clocking

Edge-based clocking may seem very simple, but do not underestimate it! You may quickly find that you did not really understand what was happening, so be sure to pay attention to this section.

Edge-based clocking is similar to state machines. You use Flip Flops (usually the D-type) to buffer data. This logic will be in the form:

```
always @(posedge clk) begin
  edgeReg1 <= someInput1;
  edgeReg2 <= someInput2 | edgeReg1;
end
```

First, note that we only include “posedge clk” in the always trigger list. This means that we will only update edgeReg1_e and edgeReg2 during the instant where clk (the clock) goes from 0 to 1. Also note that edgeReg2 will use the *previous cycle’s value* of edgeReg1_e, rather than the new value computed in the line above. Remember that Flip Flops require a setup time, and since the new value of edgeReg1_e is computed *after* the clk transition, the value saved into edgeReg2 at the clk transition must be based on the old value of edgeReg1.

When you are working on this lab, it will help you to draw timing diagrams. Make sure you understand when your edge-based and non-edge-based logic will be at each cycle. When

using edge based logic in timing diagrams, it helps to understand that your new edge-based logic will be based on the values *before* the clock transition. Also, be very careful when mixing edge-based logic, such as edgeReg1, and non-edge based logic. Edge-based logic can only change once a clock cycle (0 to 1), while non-edge based logic, such as combinational logic, may change any point in time.

2 New Verilog Tricks

Formatting Long Numbers

You may find that you are using long numbers and they are hard to read. When you are writing a paper, you might use commas every three digits (9,235,203). In Verilog, you can use an underscore in your numbers and it will be ignored. For instance, `8'b1011_0011` is completely valid and much easier to read than `8'b10110011`.

Constants

You may find that you are using the same number in your code several times but have to change this number as you change your code. Constants are very helpful here and are easy to use. To define a constant, you include the following statement at the top of your Verilog file:

```
`define MY_CONSTANT 8'h
```

Whenever you want to use the constant, simply stick a ` before the constant's name. Otherwise, it's the same as putting the number there itself:

```
if(myReg == `MY_CONSTANT) begin …
```

3 Introducing the FPGA's RAMB16_18 Memory Block

The FPGA already comes with several memory blocks. We are going to use the RAMB16_S18 module. This is a 1Kx16bit memory block (stores 1,024 different 16-bit entries). It also includes support for 2 bits of parity memory for each of the 1,024 entries, and a "Synchronous Set/Reset". The parity bits are useful for making sure your memory is not

corrupted, which is especially important in when you are exposed to radiation or gamma rays (such as in space). The Synchronous Set/Reset is not terribly useful and only sets the output of the memory to 0, but does not actually set any memory to 0. You can find a link to Xilinx's data sheet for RAMB16_S18 module on the Resources page of the class website if you want more details.

We are going to combine several of these blocks to build a memory block that can store 32 bits of data in every entry and also support 2,048 entries. We are also going to add support for a full-memory reset. This reset will set *every* entry in the memory to 0.

You'll want to create a new project, just as we did in previous labs. Don't forget to set those pins to "float". You'll also want to download and add the "RAM_A10_D16.v" Verilog file to your project. You can find this file in the Resources section of the class website. This file interfaces with one of the RAMB16_S18 memory modules and hides the parity and synchronous set/reset support. You won't need that for this lab.

4 Testing the RAMB16_18 Memory Block

Once you have added the RAM_A10_D16 module to your project, create a testbench and try it out.

- You must set the "EN" (Enable) input to 1 if you want to use the memory. If EN equals 0, nothing will happen
- If you want to write, set WE (Write) to 1. Otherwise, you will do a read.
- ADDR (Address) is the address you want to read from or write to. ADDR is 10-bits wide, meaning that you have 1,024 addresses available to you. 10'h000 is the first address and 10'h3FF is the last address.
- If you are writing, set DI (Data In) to the value you want to write. This value is 16 bits wide
- If you are reading, DO (Data Out) will be set the value of the address you just read.

- The memory block is edge-based. This means that the memory will only perform an action when CLK (Clock) rises from 0 to 1.

In your test bench, try writing to several addresses in memory, then reading from them. Also, try setting EN to 0 and check that nothing happens.

5 Designing a Larger Memory Block

In this part of the lab, we will design a memory block named “memory_ramblock” with the following specifications:

- Each entry will be 32 bits wide
- The memory will hold 2048 entries
- The memory supports a full memory reset. After a full-memory reset, each entry in the memory block will be equal to 32’h0000_0000.

Your memory block should have the following inputs and outputs (please name them exactly this way so it’s easier for the TF’s to test your work):

- **clk** (input): This is your clock
- **reset** (input): This signal is high for one cycle when you want to do a full-memory reset
- **addr** (input): This is the address you wish to use for read and writes
- **data_in** (input): This is the data you wish to save during a write
- **write** (input): If this value is 1, you wish to do a write. If it is 0, you wish to do a read.
- **req** (input): If this value is 1, you wish to do a read or write. If it is 0, you will not do anything. Note that a reset takes priority of reads and writes. Also, a reset does not require req to equal 1.

- **data_out** (output): If you performed a read, this is the data you read from memory
- **ready** (output): This signals if the memory block will accept a command in the next cycle. If ready is 1, the memory will accept a command in the next cycle. Because reads and writes should only take *one* cycle, ready should be 1 (because you can do another read, write, or reset in the *next* cycle). You will need to set ready to 0 during a reset since resets will require *many* cycles

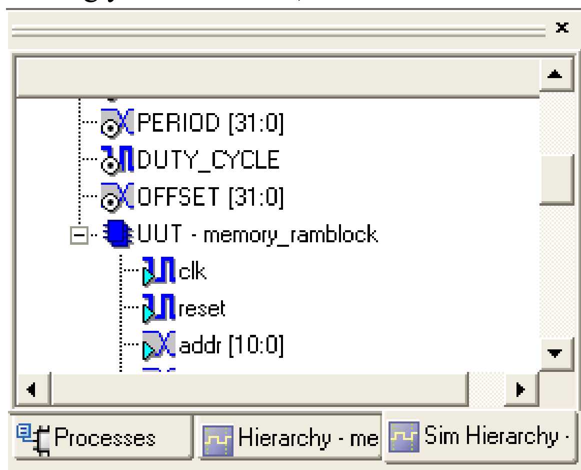
You should implement the memory block utilizing several instances of the RAM_A10_D16 modules. After you have implemented your memory block, you should write a test bench to try it out. Make sure you test reading, writing, resetting, and idling (doing nothing). There's a few to be aware of:

- Your reset may take a long time to run, since there are many entries you will have to clear. If you do a full reset in your test bench, it will have to run for a long time and will make your other tests hard to find. I would recommend using a constant to specify how many entries you want to clear, and that way you can change your code easily to make it do a partial reset:

  ```
  `define CLEAR_CYCLES ?'h3   // for quick test benches
  //`define CLEAR_CYCLES ?'h?    // for a full reset
  ```

  The above code allows you to quickly switch between a partial-reset and a full-reset. Just comment out the mode you *don't* want to use. Of course, a partial reset will not clear all of your memory, but will still show you that your reset logic is working. We will test the full-reset later, so you don't have to do it now.

You may want to peek at the contents of your memory for debugging. While you are running your test bench, look for the "Sim Hierarchy" tab and click it:



You'll notice that you can open other modules (such as the "memory_ramblock" module above) and drag these inputs into your simulation. Note that you will have to restart and re-run your simulation to see these new signals. You can use this technique to look into the data stored in each of the RAMB16_S18 blocks: the data is is stored in the "mem" array of signals.

Be aware that you must be *running* your simulation to add signals from sub-blocks. For some reason you cannot look into sub-blocks when *designing* your testbench.

6 Writing a Harness to For Onboard Testing

By this point, your memory block should be working in simulation. However, we want to make sure this works in real life too. In the last part of the lab, you will design a harness that tests your memory block on the chip and lights up LEDs so that you know your tests are passing. A harness is really another circuit specifically designed to test your memory. For your harness tests to pass, your memory block *and* your harness have to be bug free!

You will want to design your harness using edge-based clocking. This will make your work a little more complex, because you must now think about when your test inputs and outputs

will change. When working on this part, I find it very helpful to draw a timing diagram including all of your inputs and outputs, as well as noting the state of the memory at each cycle (reading, writing, resetting, idle).

I'm going to give you a skeleton of the harness:

```
module Harness(clk, status);
  input        clk;
  output [7:0] status;           // Displays test status on LEDs

  reg  [7:0] state;             // state of the harness state machine
  reg  [7:0] statusReg;       // status for LEDs

  reg        reset;
  reg  [?:0] addr;
  reg  [?:0] data_in;
  reg        write;
  reg        req;
  wire [?:0] data_out;
  wire       ready;

  assign status = statusReg;

  memory_ramblock mem(
                   .clk(clk),
                   .reset(reset),
                   .addr(addr),
                   .data_in(data_in),
                   .write(write),
                   .req(req),
                   .data_out(data_out),
                   .ready(ready)
                   );

initial begin
  statusReg <= 8'b0000_0000;
  state <= 8'd0;

  reset <= 1'b0;
  addr <= ?'h0;
  data_in <= ?'h0;
```

```
    write <= 1'b0;
    req <= 1'b0;
  end

  always @(posedge clk) begin
    case(state)
      8'd0:  begin              // Idle the first state
               state <= 8'd4;
             end

     8'd4:   begin              // Start reset
               state <= 8'd8;
               reset <= 1'b1;
             end

     8'd8:   begin              // Turn off reset, write 0x1234_5678
              reset <= 1'b0;    // to address 0x200 when mem is ready
              if(ready == 1'b1) begin
                // Light the first LED so we know the reset finished
                // successfully
                statusReg <= statusReg | 8'b0000_0001;

                state <= 8'd12;
                addr <= ?'h200;
                data_in <= ?'h1234_5678;
                write <= 1'b1;
                req <= 1'b1;
              end
             end

      8'd12: begin
      …

      default: $stop;
    endcase
   end

  endmodule
```

In the example, we start by idling for one cycle, then triggering a reset and waiting for the ready signal. We then execute a write. You will want to add your states after this logic to test writing, reading, and that a reset clears all of the memory.

You may have noticed that I tend to skip some states, jumping from state 0 to state 4 to state 8. I do this so that if I need to insert states later, we don't have to renumber the existing states or jump around. For instance, if we wanted to insert a state between state 4 and state 8, we could just create a state 6. Also, note that $stop command tells your simulator to stop simulating. This will let you use the full-reset instead of the partial-reset but still let your code stop exactly when the harness is finished.

You'll want to make sure your harness works in simulation. Create a test bench for harness. This should be fairly easy, since the only input is a clock! **Be warned, debugging your harness will probably take you the most amount of time for this lab.** You can reduce your frustration time by drawing out a timing diagram. Your friendly TFs will be happy to assist you with this and help you check for mistakes.

After you've verified that it does work, synthesize and generate your bit file, and test your design on the board. You'll want to hook your status bits up the eight LEDs on the 7 segment display (the 7 segments + the decimal point) and use CLKB, which runs at 50MHz. Because your program will run immediately after you download your code, you should quickly know if your code works on the board.

7 Hand In

When you think your code is done, please email or give your TFs a copy of your project. We have our own private harness that we will run to make sure everything is working. We also need to see your test benches and Verilog code.

8 Extra Credit

For five extra points, make your design utilize the memory at every cycle. Your previous design may make your memory run idle during a full-reset while waiting for the ready signal. Change your Verilog so that edge-based clocking logic is not forced to waste cycles idling

the memory. After you have done this, please explain what you had to change and demonstrate using your harness that the memory is not idling.

Do not try the bonus if your TF has not verified that your project works. You will not be eligible for extra credit if you do finish the regular-credit portion of the lab.

Updated November 3, 2009, Svilen Kanev