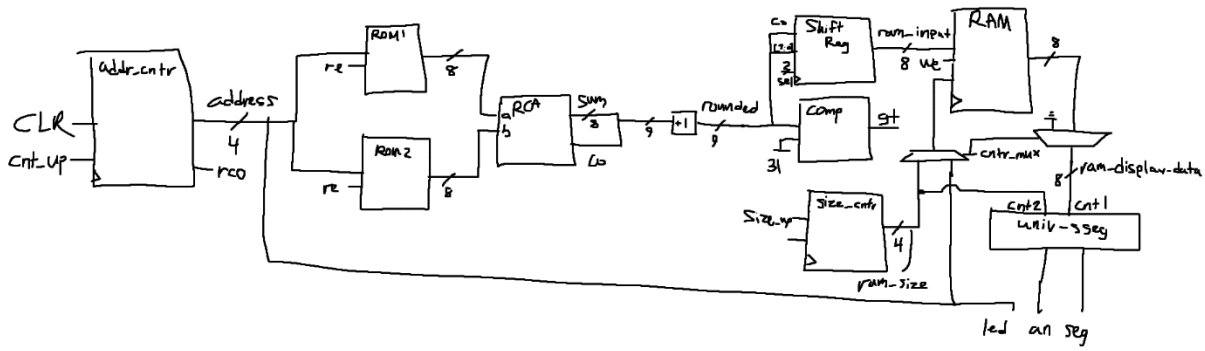
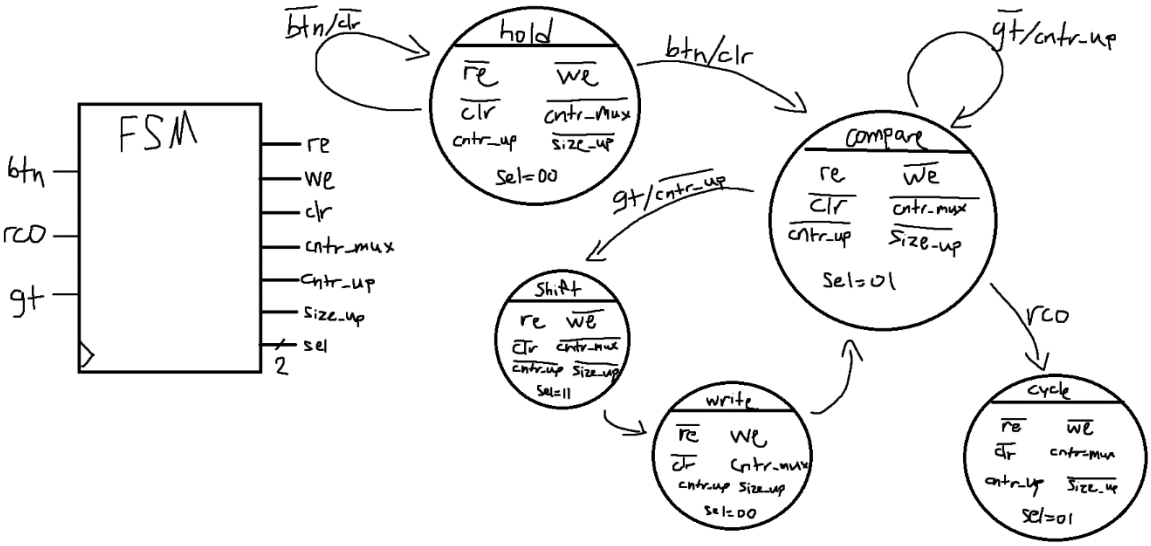
Week 1 Lab

Justin Carlson

Low Level:



FSM:



Top-Level Module:

```
module ROM_Averager(
    input clk, btn,
    output reg [3:0] address, an,
    output [7:0] seg
);

// FSM IO
logic gt; // Greater than from comparator
logic sclk; // Slower clock pulse from clock divider
logic cntr_mux; // Selector for multiplexor that handles current RAM address position
logic cntr_up; // Allows ROM address counter to increase every clock pulse
logic re; // ROM Read permissions
logic we; // RAM Write permissions
logic clr; // Clears counter when button pressed
logic size_up; // Allows RAM size counter to increment when high
logic co; // Ripple carry adder carry out
logic rco; // Counter carry out for final FSM state

logic [3:0] ram_size, ram_address; // Variables managing current size of ram and current read/write address
logic [7:0] rom_dataA, rom_dataB, rom_sum, shifted_val, ram_data, ram_display_data; // Intermediate variables for average math operations
logic [1:0] sel; // Selector for shift register that loads and shifts the rounded value to divide by two

clk_div2 clk_divider ( // Slow clock pulse to observe circuit progress
    .clk (clk),
    .sclk (sclk)   );

FSM FSM1 ( // Control circuit, variables described above
    .btn (btn),
    .gt (gt),
    .rco (rco),
    .clk (sclk),
    .cntr_mux (cntr_mux),
    .cntr_up (cntr_up),
    .re (re),
    .we (we),
    .clr (clr),
    .sel (sel),
    .size_up (size_up)   );

cntr_up_clr_nb #(.n(4)) Address_CNTR ( // Manages ROM address location and LEDs
    .clk   (sclk),
    .clr   (clr),
    .up    (cntr_up),
    .ld    (),
    .D     (),
    .count (address),
    .rco   (rco)   );

ROM_16x8a romA ( // Rom A
    .addr  (address),
    .data  (rom_dataA),
    .rd_en (re)    );
```

```
    ROM_16x8b romB ( // Rom B
        .addr  (address),
        .data  (rom_dataB),
        .rd_en (re)    );

    rca_nb #(.n(8)) RCA ( // Sums rom data together and saves carry out
        .a   (rom_dataA),
        .b   (rom_dataB),
        .cin (),
        .sum (rom_sum),
        .co  (co)    );

    comp_nb #(.n(9)) Comparator ( // Compares the rounded value to 31 to ensure it's greater than 15 before being divided
        .a  (rom_sum + 1),
        .b  (31),
        .eq (),
        .gt (gt),
        .lt ()   );

    usr_nb #(.n(8)) ShiftReg ( // Completes division operation on rounded value. Compare state in FSM loads value into module, Shift state completes division.
        .data_in (rom_sum + 1),
        .dbit (co),
        .sel (sel),
        .clk (sclk),
        .clr (),
        .data_out (shifted_val)   );

    ram_single_port #(.n(4),.m(8)) my_ram ( // RAM module that stores the average values
        .data_in  (shifted_val),  // m spec
        .addr     (ram_address),  // n spec
        .we       (we),
        .clk      (clk),
        .data_out (ram_data)   );

    univ_sseg my_univ_sseg ( // Controls output display
        .cnt1    ({6'b0, ram_display_data}),
        .cnt2    (ram_size),
        .valid   (1),
        .dp_en   (0),
        .dp_sel  (),
        .mod_sel (2'b01),
        .sign    (),
        .clk     (clk),
        .ssegs   (seg),
        .disp_en (an)   );

    cntr_up_clr_nb #(.n(4)) RAM_Size_CNTR ( // Counts the current size of the RAM module
        .clk   (sclk),
        .clr   (),
        .up    (size_up),
        .ld    (),
        .D     (),
        .count (ram_size),
        .rco   ()   );
```

```
    mux_2tl_nb  #(.n(4)) ram_display_selector_mux  ( // Inserts averages at correct RAM address then cycles in final FSM state
        .SEL    (cntr_mux),
        .D0     (ram_size),
        .D1     (address),
        .D_OUT (ram_address)   );

    mux_2tl_nb  #(.n(8)) ram_display_grounding_mux  ( // Grounds RAM output value at zero while the RAM module is being populated
        .SEL    (cntr_mux),
        .D0     (8'b0),
        .D1     (ram_data),
        .D_OUT (ram_display_data)   );

endmodule
```

FSM Code:

```
module FSM(btn, gt, rco, clk, re, we, clr, cntr_up, cntr_mux, size_up, sel);
    input btn, gt, rco, clk;
    output reg [1:0] sel;
    output reg re, we, clr, cntr_up, cntr_mux, size_up;

    //- next state & present state variables
    reg [2:0] NS, PS;
    //- bit-level state representations
    parameter [2:0] hold=3'b000, compare=3'b001, write=3'b011, cycle=3'b010, shift=3'b100;


    //- model the state registers
    always @ (posedge clk)
          PS <= NS;


    //- model the next-state and output decoders
    always @ (PS)
    begin
       re = 0; we = 0; clr = 0; cntr_up = 0; cntr_mux = 0; size_up = 0; sel = 2'b00;
       case(PS)
         hold: // Waits for user button press to begin calculations
         begin
            re = 0; we = 0; clr = 0; cntr_up = 1; cntr_mux = 0; size_up = 0; sel = 2'b00;
            if (btn == 1)
            begin
               clr = 1;
               NS = compare;
            end
            else
            begin
               NS = hold;
            end
         end

         compare: // Loads rounded ROM sum into shift register and checks if value should be stored in RAM. If rco = 1, moves to final state
            begin
               re = 1; we = 0; clr = 0; cntr_up = 0; cntr_mux = 0; size_up = 0; sel = 2'b01;
               if (rco == 1)
               begin
                   NS = cycle;
               end
               else if (gt == 1)
               begin
                   NS = shift;
               end
               else if (gt == 0)
               begin
                   cntr_up = 1;
                   NS = compare;
               end
            end
```

```
            shift: // Completes bit shift to find the average
               begin
                  re = 1; we = 1; clr = 0; cntr_up = 0; cntr_mux = 0; size_up = 0; sel = 2'b11;
                  NS = write;
               end

            write: // Writes the average value to the RAM
               begin
                  re = 1; we = 1; clr = 0; cntr_up = 1; cntr_mux = 0; size_up = 1; sel = 2'b00;
                  NS = compare;
               end

            cycle: // Cycles through the RAM contents
               begin
                  re = 0; we = 0; clr = 0; cntr_up = 1; cntr_mux = 1; size_up = 0; sel = 2'b00;
                  NS = cycle;
               end

            default: NS = hold;

            endcase
         end
      endmodule
```

Questions:

1. In your own words, provide a simple definition for digital design.

Digital design is the use of basic binary operations and concepts to change bits in a desired manner. Individual components/modules can be created with simple programming ideas and implemented to alter certain chunks of data.

2. Briefly but completely state and explain in your own words the two main aspects of the modern digital design paradigm.

Digital design can be encapsulated by two different design elements, hierarchal and modular design. Hierarchal design refers to digital modules being embedded in one another, usually for the purpose of sharing control signals and controlling its lower-level modules. Modularity refers to the use of a handful of digital modules, like registers, adders, decoders, and comparators being instantiated a number of times and used in the exact same manner to simplify the code and logic behind a circuit's operation.

3. List the accepted basic parts of a computer.

A computer is comprised of three primary components including a processor, memory, and I/O connectivity.

4. In your own words, provide a complete written description of how the circuit you designed in this experiment operates. Be sure to reference the block diagram for your circuit. This description should be more than a detailed description of the state diagram.

The ROM averaging circuit works by having a state in the FSM that waits for a user input to begin reading data. Once the circuit is told to begin, a counter starts to run on the clock cycle that provides the ROMs a memory address to read out their data. In order to calculate the average of these two ROM values, each piece of data must be sent through a ripple carry adder to sum them, round odd numbers

up to even numbers so they can evenly be divided by 2, then the rounded sum is loaded into a shift register and shifted to the right once. Now that the average of the ROM is determined, it can be compared to our value of 15 to see whether it should be loaded into the RAM. If it's less than 15, we simply ignore the value then count up to the next ROM address, but if it's greater, we want to store this value into the RAM. Once the 16 ROM addresses have been iterated through, we force the display to show the RAM module size and also cycle and output all of the values stored inside.

5. Based on your circuit design in this experiment, can you determine how many clock cycles after the button press your design requires to complete the assigned task? If so, how many clock cycles does your design require to generate the requested result? If not, explain why it is not possible to calculate the number of clock cycles.

Since there are 12 values that occupy the RAM and the clock must tick 3 times to transition between the 3 write states in my design, reading and writing the RAM values takes 36 of the slower clock ticks. There are 4 values that do not occupy the RAM, and the FSM can verify that in a single state, so 36 + 4 = 40 total slow clock pulses to fill the RAM module.

6. Moore-type FSMs are often considered "slower" than Mealy-type FSMs. Briefly explain what the notion of "slow" refers to in this case and briefly explain why Moore-types FSMs are "slower".

   Mealy-type FSMs are faster than Moore, because a Mealy output is changed before the state changes, and does not need to be reassigned in the state register on the new clock cycle. It takes a Moore-type FSM to reassign it and recognize the conditional change before it can conditionally run other code.

7. If Moore-type FSMs as slower, briefly explain why all FSMs design are not Mealy-type FSMs.

Moore-types are necessary when you need to wait a clock cycle before executing an operation. In my FSM, I needed two states to load and then shift the rounded sum in the shift register, and if I used a Mealy FSM, I would have changed my shift register selector to shift before the value was loaded (which would cause it to fail). In essence, some problems can be solved by holding an input and waiting for a state transition to execute a desired operation.

8. Describe whether you implemented your circuit using a Mealy or Moore-type FSM. Briefly state the reasons why you choose one FSM model over the other.

In my FSM, I only used one Mealy output to clear the counter once the button was pressed, and I didn't want to use a Moore output in the next state because it would clear the counter every clock cycle. The rest of my outputs were Moore outputs, because I found them simpler to work with and necessary to solve the problem with my shift register, like stated in 7. Both types can be effective for any design, and its trade-offs should be evaluated.

9. Briefly explain whether a Mealy-type output can act like a Moore-type or Mealy-type output.

A Mealy can act like a Mealy and a Moore, because it can be reassigned both on the state change, and also when a condition is met.

10. Briefly explain whether a Moore-type output can act like a Moore-type or Mealy-type output.

A Moore can only act like a Moore, because if it was changed in a state with an if condition, then it would literally be a Mealy type instead.

11. Briefly but completely explain why it is “really hard” to see the outputs of the universal seven-segment display module on the simulator’s timing diagram output even though the module has a valid data input.

I’m not entirely sure if this is the case, but since the anodes cannot all be on at the same time, they require four different clock pulses repeating themselves to drive the display. Since the data in the simulation can only read value by value, you are only seeing the value of one of the anodes at any given time.

12. Briefly explain the advantage of making an FSM, such as the one you used in this experiment, independent of external factors such as how much data the design will process.

Since the FSM is flexible with managing the size of data blocks, the RAM/ROMs could have been made to any size and worked perfectly, as the state changes apply for a ROM with 4 addresses or 128 addresses for example. The FSM simply reads from the comparator and the counter and will work until all of the addresses have been read.

13. I simplified my design such that the included FSM contains one less state than the previous design iteration. Briefly explain whether I reduced the memory requirements for the design or not.

Yes, by lowering the number of state changes, you are minimizing the amount of data that needs to be viewed and rewritten every clock cycle, and a smaller FSM leads to a smaller resource footprint for your computer.

14. Did the universal seven-segment display device used in this experiment employ lead zero blanking or not? Briefly but completely answer this question in such a way as I know that you know what lead zero blanking is.

Yes, the module did use lead zero blanking, as when it displayed the size of the RAM contents on the right two anodes, the display kept the left anode blank when the size of the RAM was in the single digits, and only turned on the second anode when it reached 10.

NOTE: I demoed my project to you in class already, and you already signed off on it. If you would like a video demo, just let me know and I can make one. Thank you!