CSM152A Lab 3: Stopwatch

Jonathan Hurwitz 804258351
Ram Sivasundaram 704261325

Lab Section 1

November 10, 2015

I. Introduction and Requirements

This lab assignment involved the full FPGA design process for a minute and second stopwatch, from preliminary testing with the simulator to UCF and final implementation in hardware. The stopwatch inputs used were the press buttons and the slider switches. The seven segment display was used to display the minutes and seconds.

The spec listed several requirements (input and outputs) for the implementation. These included:

1) An input **ADJ** to set the clock into adjustment mode. When the switch is on, the portion of the clock denoted by the selector (either minutes or seconds) will increment at a rate of 2 ticks per second (2 Hz) while the other portion of the clock will be paused.
2) An input **SEL** to select whether the minutes or seconds will be incremented at the 2Hz rate while in adjustment mode.
3) An input **RESET** to force the state machine back to state 0, thus setting the clock output to 00:00.
4) An input **PAUSE** to pause the counter while the display still shows the value that is being held.

II. Design Description

The design focused on using functional modules to implement core features, such as the counter, clock, and seven segment display. We used an iterative design process where we first designed the counters, then the clock, and then the seven-segment display. All of these modules' individual functionality was brought together in the top module to create the final product.

A. Clock Module

The term "clock" in digital logic is formally defined as a reference signal oscillating between a high and a low state. Most clocks use a 50% duty cycle but this is not required. We began by initially creating a 1Hz (for the incrementing) and a 2Hz(for incrementing in adjust mode) clock with a 50% duty cycle. However, based on the way we created our counter on the top module, we needed to change the signals to be pulse based.

```
always@(posedge clk)
begin

     if(PAUSE) //pause
       begin
       end
     else if(RESET) //reset
        begin
            zero <= 0;
            one  <= 0;
            two  <= 0;
            three <= 0;
        end
     else if(!ADJ && CLK1)
       begin
         //Led0 <= 0;
          $display("%d"+"%d"+"%d"+"%d", three, two, one, zero);
          if(zero == 9)
              begin
                  if(one == 5)
                      begin
                          one <= 0;
                          zero <= 0;
                          if(two == 9)
                              begin
                                  if(three == 5)
                                      begin
                                          three <= 0;

                                      end
                                  else
                                      begin
                                          three <= three+1;
                                      end
                                      two <= 0;
                              end
                          else
                              begin
                                  two <= two+1;
                              end
                      end
                  else
                      begin
                          zero <= 0;
                          one <= one+1;
                      end
              end
          else
              begin
                  zero <= zero+1;
              end
          end
```

Figure 1: Top module case-based counter implementation.

The above figure shows the top module implementation of a case-based counter by check BCD values for edge cases and taking care of the specific circumstances, such as incrementing the next-most significant digit and setting the current digit back to zero. The block of code shown above has a check for a CLK1 high in order to handle the case where adjust is off. CLK1 was changed from a 50% duty cycle clock into this pulse because this check would execute many consecutive times rather than just on one period of the 1Hz clock because the always@ block is triggering on the 100MHz reference clock. A similar approach was taken to create the adjust mode case, which is shown in the figure below.

```
        if(ADJ && CLK2)
          begin
          $display("%d"+"%d"+"%d"+"%d", three, two, one, zero);
            //Led0 <= 1;
            if(SEL == 1) //seconds
                begin
                 // Led1 <= 1;

                    if(zero == 9)
                        begin
                            if(one == 5 )
                                begin
                                    zero <= 0;
                                    one <= 0;
                                end
                            else
                                begin
                                    one <= one + 1;
                                    zero <= 0;
                                end
                        end
                     else
                        begin
                            zero <= zero + 1;
                        end
                end
            else if(SEL == 0) //minutes
                begin
                    if(two == 9)
                        begin
                            if(three == 5)
                                begin
                                    two <= 0;
                                    three <= 0;
                                end
                            else
                                begin
                                    three <= three + 1;
                                    two <= 0;
                                end
                        end
                     else
                        begin
                            two <= two + 1;
                        end
                end
          end

     end
    endmodule
```

Figure 2: Top module case-based counter implementation showing the second part of the always@ block, where the ADJ HIGH case is taken into account.

The same thing was done to take care of the 2Hz incrementing when ADJ is logic HIGH. The 2Hz signal was a pulse rather than a 50% duty cycle clock. The inputs and outputs are:

1) input CLK_REF: This is the 100MHz reference clock from the FPGA fed in from the top module.
2) input CLK_RES: This is a reset signal fed in from the top module. It sets the state of both CLK1 and CLK2 to be low and sets the counter equal to zero.
3) output reg CLK_2HZ: This is the CLK2, 2Hz pulse signal used for incrementing when the ADJ value is high.
4) output reg CLK_1HZ: This is the CLK1, 1Hz pulse signal used for incrementing when the ADJ value is low.

The code for the clock module included in ”clock.v” is shown below.

```
module clock(
    input CLK_REF,
    input CLK_RES,
    output reg CLK_2HZ,
    output reg CLK_1HZ
    );


reg [26:0]count1 = 0;

/* 1Hz and 2Hz outputs*/
always@(posedge CLK_REF)
begin
   if(CLK_RES)
      begin
         count1[26:0] <= 20'h00000;
         CLK_1HZ <= 1'b0;
         CLK_2HZ <= 1'b0;
      end
   else
      begin
          if(count1[26:0] == 50000000) //2Hz
            begin
               CLK_2HZ <= 1;
               count1 <= count1 +1;
            end
         else if(count1[26:0] == 100000000) //1Hz
            begin
               CLK_1HZ <= 1;
               CLK_2HZ <= 1;
               count1 <= 20'h00000;
            end
         else
            begin
               CLK_2HZ <= 0;
               CLK_1HZ <= 0;
               count1 <= count1 + 1;
            end
      end
end
endmodule
```

Figure 3: Code block showing the clock module implementation with the pulse-based counters.

Our module requires only one counter, and we rely on clock division of the 100MHz clock to check for specific cases in order to set the pulses equal to HIGH for one cycle of the 100MHz clock.

B. Display Module

The seven-segment display can only display one number between zero and nine across all digits at one time, so multiplexing the digits at a relatively high frequency is required to create the illusion of digit stability and make it appear as if the digits are different.

The module inputs and outputs are:

1) input CLK: This is the 100MHz reference clock from the FPGA used for edge triggering.
2) input CLK1: 1Hz clock input.
3) input CLK2: 2Hz clock input.
4) input ADJ: Adjustment input from top module.
5) input SEL: Selector input to be used when ADJ is HIGH.
6) input RESET: The reset signal comes from the user pressing the reset button. This will set the digit to display equal to zero from within this module and for the duration of the reset signal display a blank value.
7) input [3:0] d0, d1, d2, d3: These 4-bit regs correspond to the zero, one, two, and three digit positions from the

top module. The multiplexed digit will be set to one of these four values depending on the state.

8) output reg[6:0] dispDigit: This 7-bit reg includes all of the values to display a digit on the seven segment display.
9) output reg[3:0] selector: The selector determines which of the digits will be selected to display.

We moved the fast clock into the display module since it was counter based and it made more sense to make it local.

```
module disp(
    input CLK,
    input CLK1,
    input CLK2,
    input RESET,
    input ADJ,
    input SEL,
    input [3:0] d0,
    input [3:0] d1,
    input [3:0] d2,
    input [3:0] d3,

    output reg[6:0] dispDigit,
    output reg[3:0] selector
    );
/*
Selector reg determines which of the digits will be selected.
dispDigit holds information regarding which segments of the 7 to illuminate.
*/
reg [3:0] mDigit; //the digit we are currently using
localparam SIZE = 14;
localparam B_SIZE = 24;
reg [SIZE-1:0]count;
reg [B_SIZE-1:0] bcount;

 always@(posedge CLK)
 begin
   if(RESET)
      begin
         count <= 0;

         //CLK_FAST <= 1'b0;
      end
   else
      begin
         if(count[SIZE-1:SIZE-3] == 7)
            begin
               //CLK_FAST <= ~CLK_FAST;
               count <= 0;
            end
         else
            begin
               count <= count + 1;
            end
      end
 end
```

Figure 4: Local implementation of the fast clock used for display multiplexing.

The counter variable used in this clock was also used to decide which digit to select in the multiplexer. Bits MSB and MSB-1 can be viewed as providing four possible cases: 00, 01, 10, 11. These cases were checked for on each rising edge of the 100MHz clock and a corresponding digit was selected.

```
always@(posedge CLK)
begin
   if(RESET)
      begin
         //non displayable value
         selector = 4'b1111;
         mDigit = 0;
      end
   else
      begin
         case(count[SIZE-1:SIZE-2])
            0: //EN digit 0
               begin
                  selector = 4'b1110;
                  mDigit = d0;
               end
            1: //EN digit 1
               begin
                  selector = 4'b1101;
                  mDigit = d1;
               end
            2: //EN digit 2
               begin
                  selector = 4'b1011;
                  mDigit = d2;
               end
            3: //EN digit 3
               begin
                  selector = 4'b0111;
                  mDigit = d3;
               end
            default:
               begin
                  mDigit = 0;
               end
         endcase
      end
end
```

Figure 5: Using the fast clock's counter MSB and MSB-1 as the conditions for changing state.

Finally, whenever the multiplexed digit "mDigit" was changed, the value of dispDigit was updated with the new value. This is shown in the image below:

```
/*
Case covering possible values of a 4-bit reg.
7'bgfedcba

We want to change the display digit every time
the mDigit (multiplexed digit) changes.
*/
always@(mDigit)
   if(RESET)
      begin
      dispDigit = 7'b1000000;
      end
   else
      begin
            case (mDigit)
                4'b0000 : dispDigit = 7'b1000000;   // 0
                4'b0001 : dispDigit = 7'b1111001;   // 1
                4'b0010 : dispDigit = 7'b0100100;   // 2
                4'b0011 : dispDigit = 7'b0110000;   // 3
                4'b0100 : dispDigit = 7'b0011001;   // 4
                4'b0101 : dispDigit = 7'b0010010;   // 5
                4'b0110 : dispDigit = 7'b0000010;   // 6
                4'b0111 : dispDigit = 7'b1111000;   // 7
                4'b1000 : dispDigit = 7'b0000000;   // 8
                4'b1001 : dispDigit = 7'b0010000;   // 9
                default : dispDigit = 7'b0111111;   // dash
            endcase
      end
endmodule
```

Figure 6: Switching on the value of mDigit to update dispDigit accordingly. Recall dispDigit is the 7-bit output that corresponds to which parts of the display to light up.

By tweaking the value of the localparam SIZE, we were able to change the speed of the clock. Smaller values SIZE will mean that the cases are reached more quickly, whereas larger values will make the clock slower.

The blinking was triggered when ADJ was set to HIGH. The selector switch SEL denoted whether minutes or seconds would blink. To make the blinking, we held the opposite two digits at the high frequency of ran the digits of interest at a

lower frequency. This code took many lines and can be viewed in the source code for ”disp.v.”

C. Counter Module

We initially created mod-6 and mod-10 counters in a file called ”counter.v” for use in providing the digit counting functionality. However, we decided that it was simpler to check BCD states in the top module and handle special cases (when fives and nines occur in upper and lower digits respectively.) This resulted in an easy to implement solution but it isn't very clean and takes up many lines.

D. Stopwatch Module (Top Module)

The top module ”SW.v” brings together all of the submodules and implements both the adjustment and selector features. The inputs and outputs are defined as follows:

1) input clk: 100MHz reference clock from the FPGA. This gets passed into the instantiation of the clock module as well as the display module.
2) input RESET: Reset signal coming from a button specified in the ucf file. Passed into both sub-modules.
3) input PAUSE: Pause signal coming from a button specified in the ucf file. Passed into both sub-modules.
4) input ADJ: The adjustment signal that denotes whether or not the clock should go into adjust mode. This also comes from a button specified in the ucf file.
5) input SEL: The selector signal that specifies the which set of digits to increment when ADJ is set to HIGH.
6) output[6:0] dispDigit: 7-bit output reg that is updated by the display module.
7) output[3:0] selector: 4-bit output reg that is updated by the display module and denotes how to light up digits.

Figure 2. shows the second part of the top module's always@ block. This portion deals with the case where ADJ is HIGH. The clock should be incremented at 2Hz, so there is also an AND check for CLK2 being HIGH as well. The cases for SEL = 1 or SEL = 0 include incrementing for either the zero or the two digit respectively.

Adding this to the top module was not the cleanest way, because code was duplicated and we never ended up using our counter module.

```
## Clock signal
NET "clk"             LOC = "V10"; #| IOSTANDARD = "LVCMOS33";   #Bank = 2, pin name = IO_L30N_GCLK0_USERCCLK,
#Net "clk" TNM_NET = sys_clk_pin;
#TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;



## 7 segment display
NET "dispDigit[0]"          LOC = "T17"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L51P_M1DQ12,
NET "dispDigit[1]"          LOC = "T18"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L51N_M1DQ13,
NET "dispDigit[2]"          LOC = "U17"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L52P_M1DQ14,
NET "dispDigit[3]"          LOC = "U18"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L52N_M1DQ15,
NET "dispDigit[4]"          LOC = "M14"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L53P,
NET "dispDigit[5]"          LOC = "N14"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L53N_VREF,
NET "dispDigit[6]"          LOC = "L14"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L61P,
#NET "dispDigit[7]"          LOC = "M13" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L61N,

NET "selector[0]"           LOC = "N16"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L50N_M1UDQSN,
NET "selector[1]"           LOC = "N15"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L50P_M1UDQS,
NET "selector[2]"           LOC = "P18"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L49N_M1DQ11,
NET "selector[3]"           LOC = "P17"; #| IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L49P_M1DQ10,


## Switches
NET "SEL"           LOC = "T10" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L29N_GCLK2,
NET "SEL"           LOC = "T9"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L32P_GCLK29,
NET "SEL"           LOC = "V9"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L32N_GCLK28,
NET "SEL"           LOC = "M8"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L40P,
NET "SEL"           LOC = "N8"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L40N,
NET "SEL"           LOC = "U8"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L41P,
NET "SEL"           LOC = "V8"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L41N_VREF,
NET "SEL"           LOC = "T5"  | IOSTANDARD = "LVCMOS33";   #Bank = MISC, Pin name = IO_L48N_RDWR_B_VREF_2,


## Button s
NET "ADJ"         LOC = "B8";  #| IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name = IO_L33P,
NET "ADJ"         LOC = "A8";  #| IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name = IO_L33N,
NET "RESET"         LOC = "C4";  #| IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name = IO_L1N_VREF,
NET "PAUSE"         LOC = "C9";  #pause| IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name = IO_L34N_GCLK18,
NET "ADJ"         LOC = "D9";  #reset| IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name = IO_L34P_GCLK19,
```

Figure 7: The UCF file specifying how pins were connected to inputs and outputs in the modules.

The UCF file from lab 1 was modified and changed to suit our design. We noticed through iterative testing that certain buttons and slider switches were more receptive of assignments than others, so the settings displayed above were the optimal ones.

III. Simulation Documentation

We used the simulator to test basic functionality of our counter and clocks. We checked to make sure that the counters were setting that carry out and main regs accordingly at their respective mod values, despite not using the counter in our final implementation. We verified that the clock pulses were appearing at the right time stamps.

Rather than looking at solely waveforms, we added $display statements in the top module where the counter is implemented in order to see the minutes and seconds increment. Since the simulator takes a long time to simulate on the order of seconds, we triggered each counter on the 100MHz clock to speed things up. This allowed us to verify the carry behavior for cases such as xx:x9, x9:x9, x9:59, and 59:59.

Once we confirmed that everything was working as expected, we created a programming file and tested on the FPGA. The only difficulty that came with working with the hardware was getting the buttons to work properly. Some buttons on the nexsys3 board just weren't receptive of input.

For reference, the test bench files are included with the project zip file turned in.

IV. Conclusion

The goal of this project was to create a modular solution to a simple problem that required careful consideration of boundary cases and timing. The trials encountered in this lab further reinforced the idea that timing is everything with digital logic. Clock errors and triggering on the wrong clock caused us issues with the counters. This was fixed by looking at waveforms and time stamps in the simulator.

One difficulty was implementing blinking. This required holding two digits constant while the others blinked and the LSB of the subset incremented at 2Hz. We needed to add a separate counter ”bcount” withint the display module in order to create different updating speeds in the case statement.

The process to go from code to hardware was very straightforward. The UCF file allows pin designators to be tied to the inputs and outputs of the top module, so we just renamed the specific pins we needed to use.

Our design could be improved in the future by using the counter module rather than having it integrated into the top module.