CSM152A
Team 4 (Justin and Krish)
Date: Mar 3, 2024

Lab Report 3: Stopwatch on the Basys3 Module

Introduction:

This project centers around the development of a stopwatch module using the seven segment LED display, with additional functionalities such as an adjustment mode, a reset button, and pause button on a basys3 FPGA board. Using the FPGA's 100 MHz clock that drives its operations, we create 4 internal clocks (a 1 Hz clock to update the seconds, a 2Hz clock to update the minutes/seconds for adjustment mode, and 4 Hz clock for blinking, and a 381 Hz clock to update the values on the seven segment display). The 7 seven segment display allows us to display the minutes and seconds, where the rightmost 2 digits show the seconds count and the other to display the minutes count. Using these helper clock functions and logic diagrams given below in the design section, we recreated the stopwatch module in a robust manner. Along with that, we used debouncing to handle the noise from using flipflops and switches as inputs and prevent us from getting unpredictable results when resetting or pausing the stopwatch. Our stopwatch also accommodates an adjustment mode, which is set by the first flip flop (SW 0). When this switch would be set to high, it would allow either the minutes or the seconds to increment by two ticks per second. The section of the stopwatch that would be incremented in adjust mode is determined using the second flip flop(SW 1) and the field that is currently being adjusted would blink at a rate greater than 2 hertz even when paused.

Design:

Our design is made up of a singular module named stopwatch. This module uses the 100 Mhz clock (clk) of the FPGA board to create a working stopwatch which outputs the time in minutes and seconds on the board's 7-segment display. The module also takes input from the center button (used for pausing the stopwatch), the right button (used for resetting the stopwatch), and two switches (used for putting the stopwatch in adjustment mode and selecting whether minutes or seconds should be adjusted). Whilst in adjustment mode, depending on whether minutes or seconds is selected, that portion of the stopwatch will blink and increment at a frequency of 2 hz, while the other portion stays frozen.

Debouncing

To start, the stopwatch module debounces all of the buttons and switches which are used as input. This was done using the same method used in Lab 1. For our reset button, we assigned its value to a wire (arst_i), and then set up a 2-bit register (arst_ff) to be set only on the positive edge of clk or arst_i. When arst_ff is being set, if arst_i is high then arst_ff is set to 11, otherwise the register is bit shifted once to the right. We then assign the value of the lowest bit of arst_ff (arst_ff[0]) to a wire named rst which we use to control our resetting. This prevents any unstable inputs, because it prevents the value of rst from changing too quickly. The value of the apause_ff register was set using this same method as well. For our switches, we store their value in a register (_sw) only when a 381 hz clock (which we implement in a later section) is set to high. This similarly ensures that the value we read from the switches does not change too rapidly.

```
assign arst_i = btnR;
assign rst = arst_ff[0];

always @ (posedge clk or posedge arst_i)
        if (arst_i)
          arst_ff <= 2'b11;
        else
          arst_ff <= {1'b0, arst_ff[1]};
```
Code for debouncing reset button

Pause Toggling
Unlike the reset button, our pause button needed to toggle whether the stopwatch was paused or not. Thus, instead of assigning the value of apause_ff[0] to a wire directly, we instead have a register named pause which has its value flipped on the positive edge of apause_ff[0]. This both allows for proper toggling functionality and prevents rapid toggling when holding down the pause button.

Clock Dividers
For our design, we needed several different clocks with various timings which were set up by again using a similar method to Lab 1.

We first implemented a 381 hz clock which would be used for updating the display and the values of the switches. This was done by assigning a 19-bit wire (clk_dv_inc) to be the value of the 18-bit register (clk_dv) plus one. On the positive edge of clk, we set the value of the clk_dv register to be the lower 18 bits of clk_dv_inc. This essentially causes clk_dv_inc to be incremented once every positive edge of clk and be reset to 0 after its 19th bit is first set. The 19th bit of clk_dv_inc is set to high with our needed timing, so we use this as our 381 hz clock. Finally, in order to use the 381 hz clock for display we set the value of a 2-bit register (clk_display) on the positive edge of clk to be incremented by the value of the 19th bit of clk_dv_inc. This makes clk_display cycle through the values 0-3 on a 381 hz frequency.

```
assign clk_dv_inc = clk_dv + 1;
assign clk_blink_inc = clk_blink_dv + 1;


always @ (posedge clk)
     clk_dv   <= clk_dv_inc[17:0];
     clk_display   <= clk_display + clk_dv_inc[18]; //381 Hz
```
Code used for incrementing clk_display

For the clock which we used to control blinking, we used a similar method, however for this clock we needed it to be high half the time and low the other half. Using a 24-bit wire (clk_blink_inc) and register (clk_blink_dv), we followed the same process as above, but simply set the value of clk_blink_dv to the full value of clk_blink_inc on the positive edge of clock instead. Thus, the highest bit of clk_blink_inc would stay high for half the time and reset when it overflowed.

Finally we implemented a 1 hz clock (used for incrementing the stopwatch) and a 2 hz clock (used for incrementing the portion of the stopwatch in adjustment mode). For each of these we set up an integer (clk_1hz_inc and clk_2hz_inc) which would be set on the positive edge of clk to be its current value incremented by one modulo a certain number. For the 1 hz clock this number was 100000000 and for the 2 hz clock it was 50000000. From this implementation, both clk_1hz_inc and clk_2hz_inc would be set to 0 on the frequencies we needed.

```
clk_2hz_inc = (clk_2hz_inc + 1) % 50000000;
clk_1hz_inc = (clk_1hz_inc + 1) % 100000000;
```
Code for 1 hz and 2 hz clocks

Stopwatch Time

For this design, we needed a value to store the current time which should be displayed on the stopwatch and increment it accordingly. We did this by using an integer called clk_val which stores the current time in number of seconds. The value of clk_val was initialized to 0, so the stopwatch would always start at 00:00. If rst was high on the positive edge of clk, then clk_val would be set to 0, creating our reset functionality. For incrementing clk_val we had to perform some additional logic. First, if the pause wire was high, then clk_val would not increment at all. Otherwise, if the value of our adjustment switch (_sw[0]) was low and clk_1hz_inc == 0, then clk_val would be incremented by one, which was the standard timing for the stopwatch. If _sw[0] was high and clk_2hz_inc == 0, then the value of clk_val would be incremented by 1 if our selection switch (_sw[1]) was high or 60 if it was low. This created adjustment functionality for our stopwatch. In any other case, the value of clk_val would not be incremented and would stay the same.

```
if (_sw[0] && clk_2hz_inc == 0 && ~pause)
        clk_val = clk_val + (_sw[1] ? 1 : 60);

if (~_sw[0] && clk_1hz_inc == 0 && ~pause)
        clk_val = clk_val + 1;

if (rst)
        clk_val = 0;
```

Code for incrementing and resetting clk_val

Display

For controlling the 7-segment display we needed to set the value of the anode (an) and segment (seg) registers of the board. In an always @* block we set up a case statement using the value of clk_display. In the order of the cases 0-3 the value of an would be set to 0111, 1011, 1101, and finally 1110. This would allow us to control each of the 4 digits on the display depending on the value of clk_display. In this case statement we also set the value of a 4-bit register called LED_BCD which would be used for storing the number we wanted to display on a certain digit of our display. The value of LED_BCD would be set using the value in clk_val after some operations to get the proper display value depending on what digit is being set. To control the blinking, if clk_blink_inc[24] and _sw[0] were both high, then in cases 0 and 1 if _sw[1] was low, LED_BCD would be set to 1111, causing the minutes to blink. Otherwise, in cases 2 and 3 if _sw[1] was high, LED_BCD was set to the same value, causing the seconds to blink. Finally in a separate always @* block, the value of LED_BCD would be passed through a case statement used to control the value of seg. If LED_BCD was set to 1111, then the default case would occur where seg is set to display nothing.

```
always @* begin
        case(clk_display)
            2'b00: begin
                an = 4'b0111;
                LED_BCD = (clk_blink_inc[24] & _sw[0] & !_sw[1]) ? 4'b1111 : ( (clk_val / 600) % 6 );
            end
            2'b01: begin
                an = 4'b1011;
                LED_BCD = (clk_blink_inc[24] & _sw[0] & !_sw[1]) ? 4'b1111 : ( (clk_val / 60) % 10 );
            end
            2'b10: begin
            an = 4'b1101;
            LED_BCD = (clk_blink_inc[24] & _sw[0] & _sw[1])  ? 4'b1111 : ( (clk_val % 60) / 10 );
        end
        2'b11: begin
            an = 4'b1110;
            LED_BCD = (clk_blink_inc[24] & _sw[0] & _sw[1]) ? 4'b1111 : ( clk_val % 10 );
        end
    endcase
end
```

Code for setting value of 7-segment display digits. LED_BCD is converted to the proper value for the segment register in a different block.

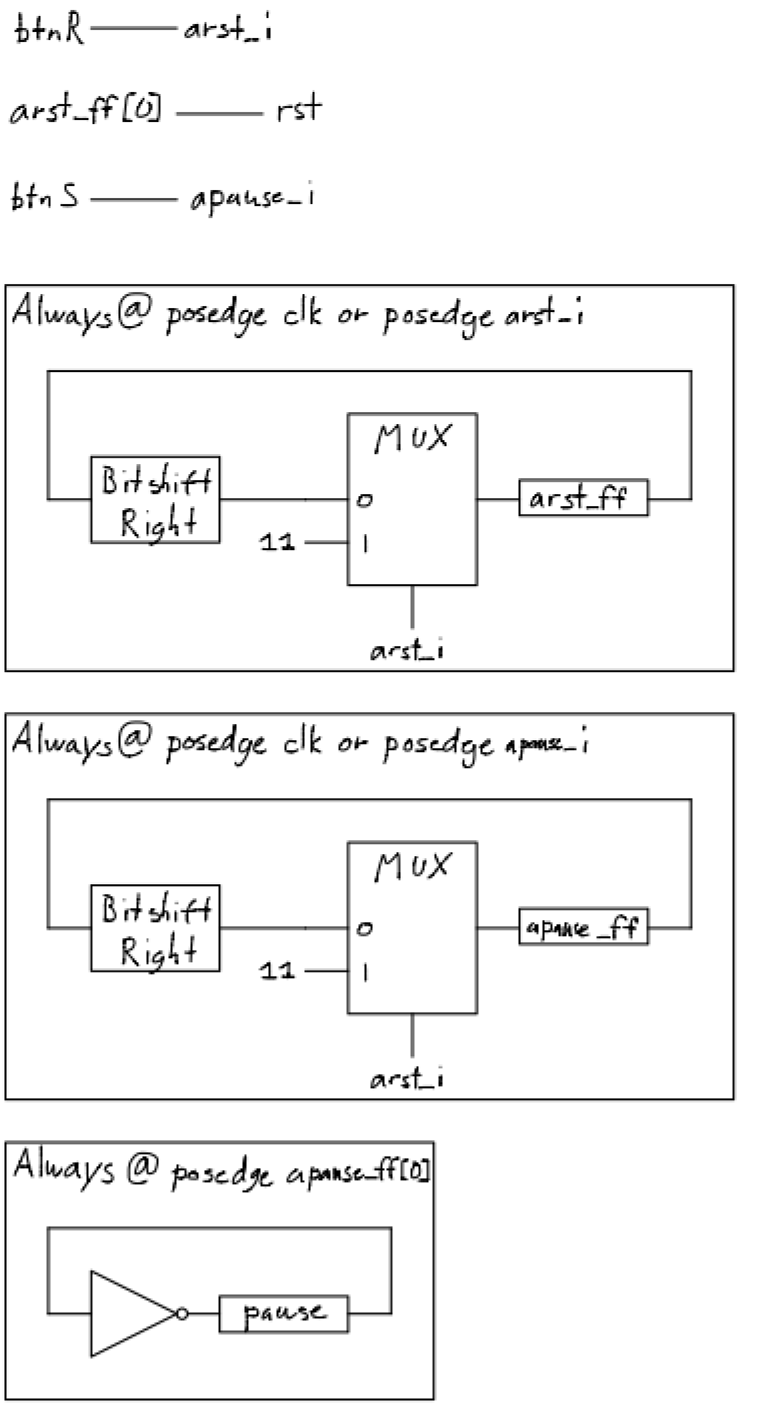
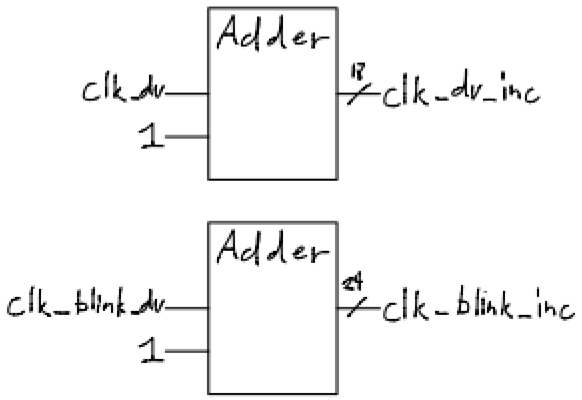
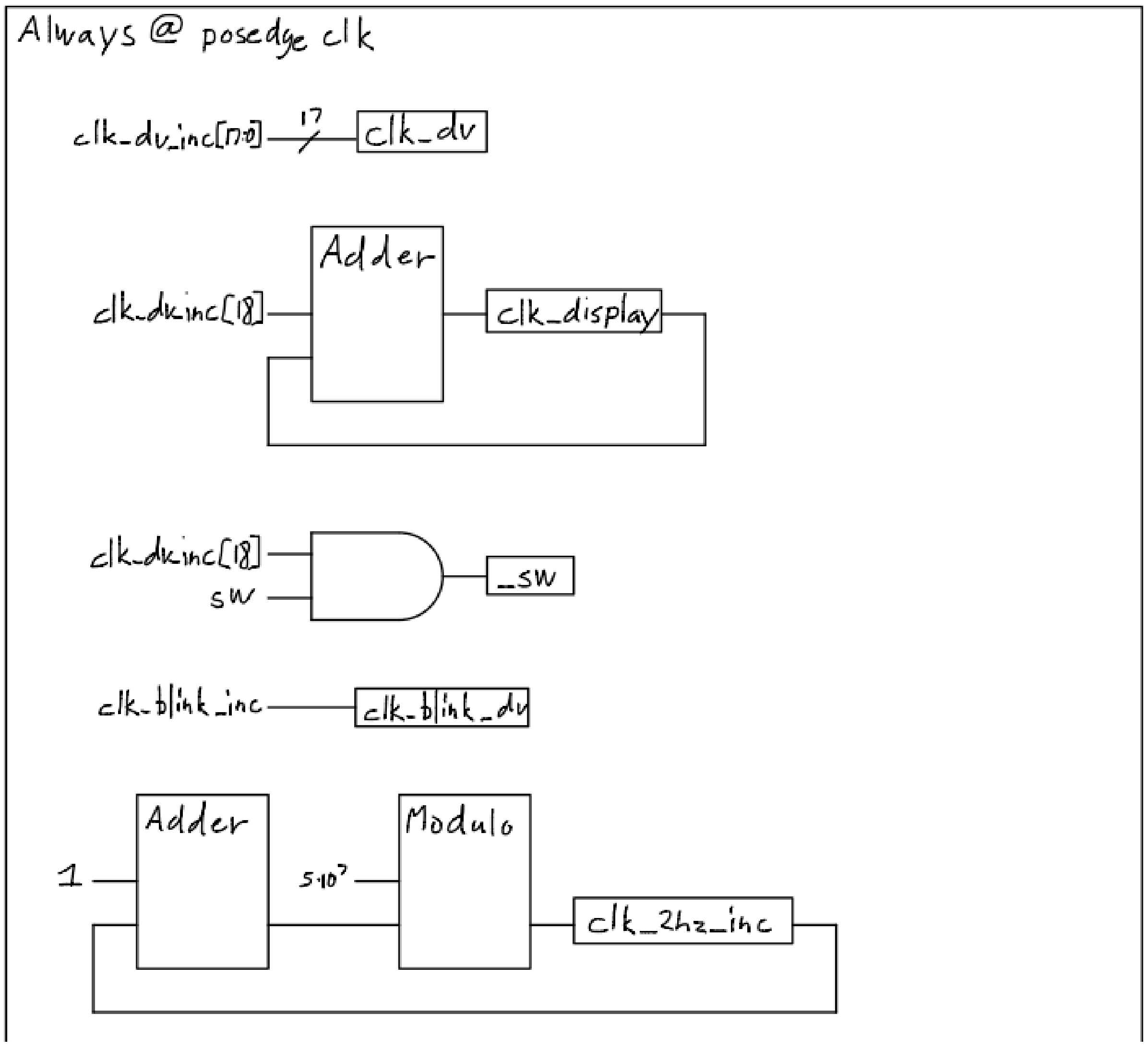


Figure 1. Logical design diagram for button debouncing


Adder
clk_dv
1
17
clk_dv_inc
Adder
clk_blink_dv
1
24
clk_blink_inc


Always @ posedge clk
clk_dv_inc[17:0]
17
clk_dv
Adder
clk_dv_inc[18]
clk_display
clk_dv_inc[18]
sw
_sw
clk_blink_inc
clk_blink_dv
Adder
Modulo
1
5·10^7
clk_2hz_inc

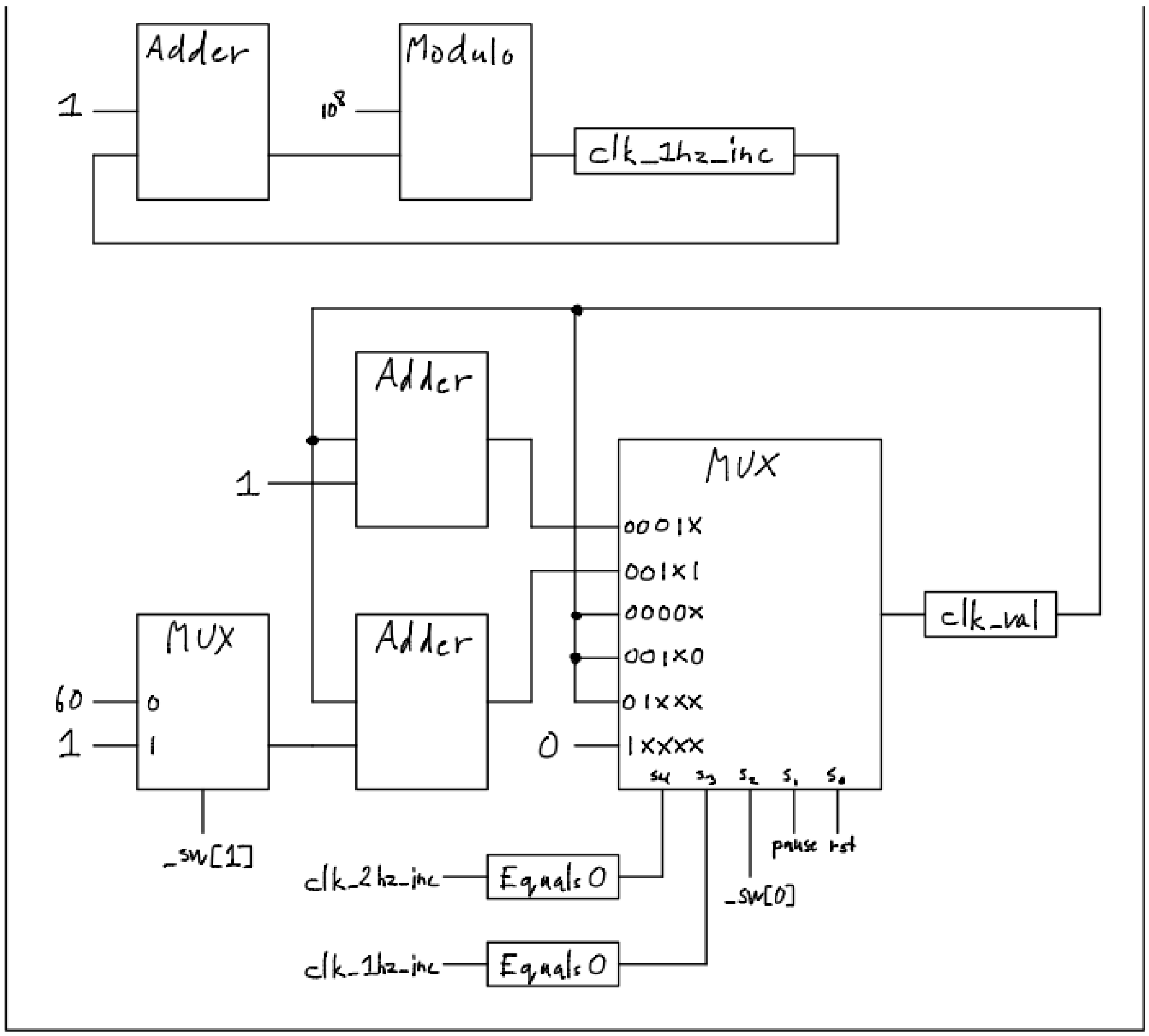


Figure 2. Logical design diagram for clock dividers and incrementation of clk_val

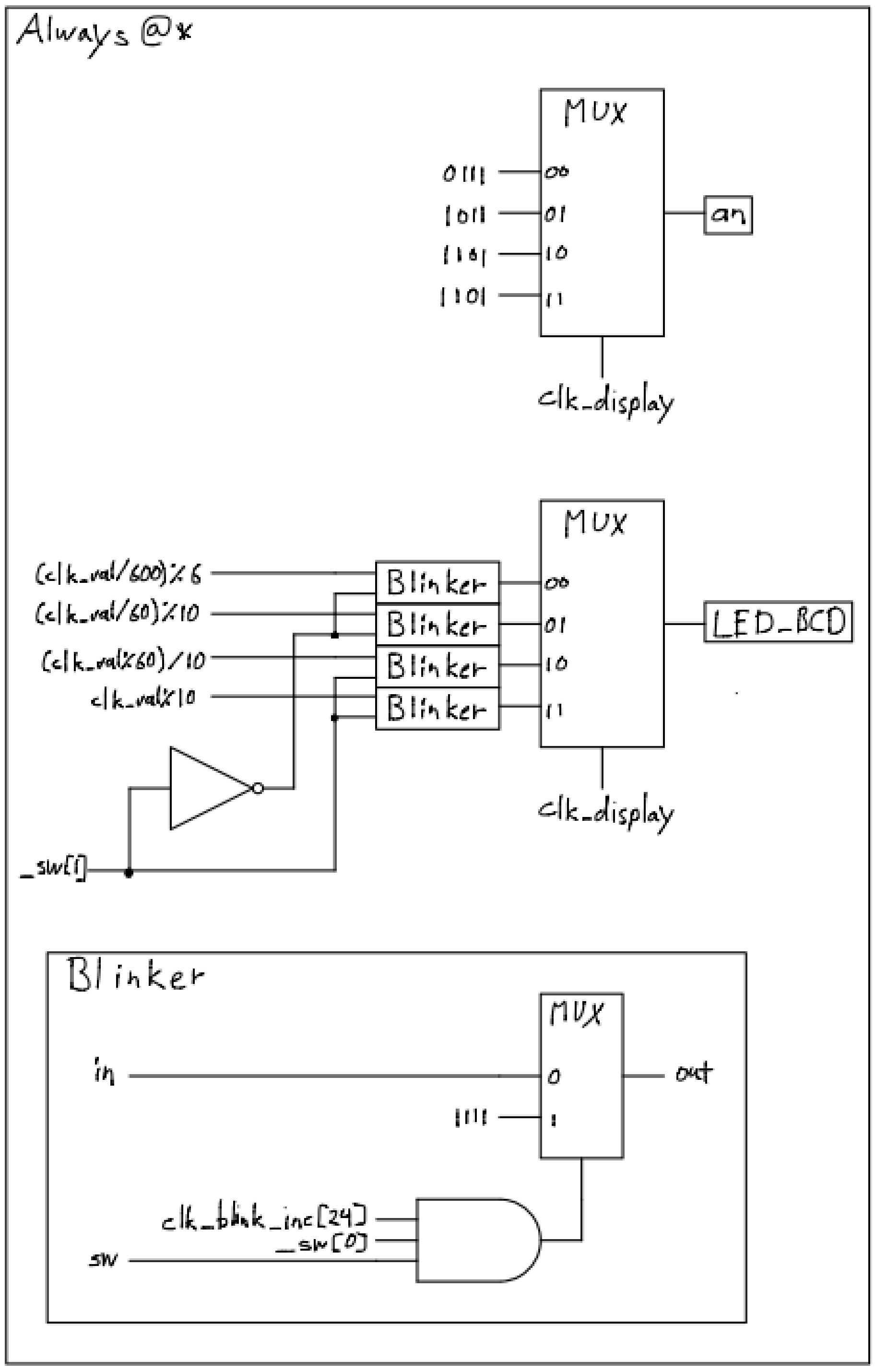


Figure 3. Logical design diagram for controlling 7-segment display

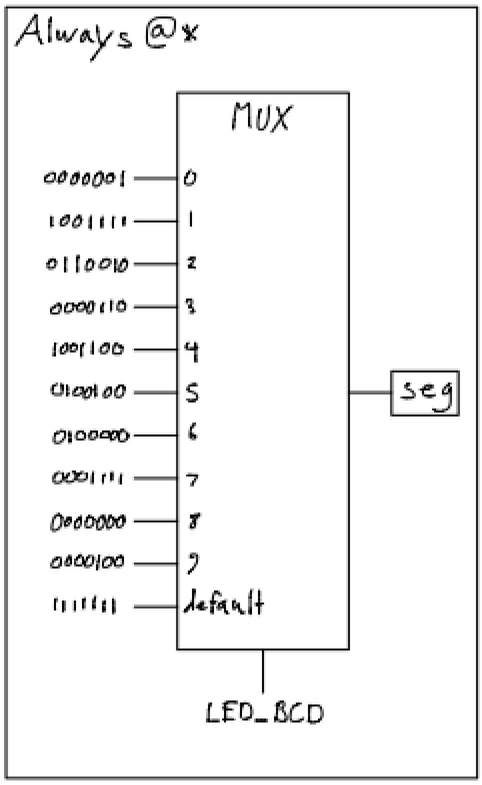


Figure 4. Logical design diagram for conversion from LED_BCD to seg

Simulation and test cases:

Test cases: To ensure that our clock worked correctly and was accurate to the seconds, we tested it against the stopwatch on our phones. We set up both stopwatches and watched them for a few minutes to see if they stayed in sync. This way, we could spot any differences between our clock and the phone's stopwatch thus checking the validity and reliability of our stopwatch. We also did some tests to check debouncing by starting and stopping for a few minutes(every alternate second) to see if our pause button wouldn't toggle correctly if mechanical bouncing would give rapid fluctuations in the signal.

We also tested out the additional features including adjusting the time, pausing the clock, and blinking the minutes/seconds section if it was being adjusted in adjustment mode. By testing these features in different situations and comparing them with our mobile phone's stopwatch, we made sure our clock was indeed updating at 2 ticks per second(averaging over a few hundred seconds) and carrying out the other functionalities with their correctly timed clocks.

Issues and Bugs Encountered:
While implementing the logic and carrying out tests, we encountered a number of issues including both logical errors and syntax errors. While looking up for modules to control the seven segment display, we were unable to figure out how to control the seven segment LED display and were only able to display a single digit in a given time frame. To solve this issue, we turned to tutorials referenced online documenting the use of LED segments and how a refresh clock would allow us to display all 4 digits correctly. After understanding our refresh clock(381 Hz), we integrated it with the display function(which uses switch-case statements to handle single digits) and got our LED display to output the correct values.

Another issue that we encountered was when we were carrying out debouncing for the pause and reset button. Unsure as to whether our buttons were carrying out debouncing correctly, we referred to the code in lab1 to see how the debouncing was being handled. We then tried manually testing the buttons to see if our debouncing section of the code was working correctly(we were mainly focused on the pause button functionality) to see if the button would cause the stopwatch to toggle the start and stop correctly.

We also faced issues with overflow when working on the overflow for the minutes section, and weren't able to handle overflow correctly when performing a modulus of the display time by 59*60+ 59(the maximum time that the clock could display in seconds), which was causing our display to show erroneous results as soon as the minutes went over 59, and was affecting the seconds digits as well. Thus, we resolved this by solving it on the display end, where we were modding the tens digit of the minutes by 6, effectively handling the overflow issue.

<u>Conclusion:</u>
In conclusion, we were able to implement the stopwatch functionalities on our FPGA board using the constraints and guidelines in Lab 3 instructions pdf. We implemented the additional functionalities such as adjustment mode, a reset button, and a pause button to enhance the user experience. By leveraging the FPGA's 100 MHz clock, we developed 4 internal clocks for various tasks, including updating seconds, managing adjustment modes, blinking functionality, and refreshing the display values. Despite facing initial challenges in understanding how to control the seven-segment LED display and effectively debounce the buttons, we were able to generate a robust stopwatch by referring to online resources and previous lab codes(lab 1), ultimately overcoming these obstacles.