UCD School of Electrical and Electronic Engineering

EEEN30190 Digital System Design

# Bicycle Light Design

|  |  |
|---|---|
| **Name: Jack Coleman** | **Student Number: 21207103** |
| **Name: Ruairi Hogan** | **Student Number: 21432816** |
| **Lab Session: Thu 10** | **Brightspace Group: 5** |

## Contents

High-Level Design ........ 2
  Features ........ 2
  User Guide ........ 3
Block Design ........ 3
  "cleanup" Module (Jack Coleman) ........ 4
  "selectMode" Module (Ruairi Hogan) ........ 8
Verification ........ 12
  "cleanup" Module (Ruairi Hogan) ........ 12
  "selectMode" Verification (Jack Coleman) ........ 13
  Complete Design Verification ........ 15
Hardware Testing ........ 16
Conclusion ........ 17

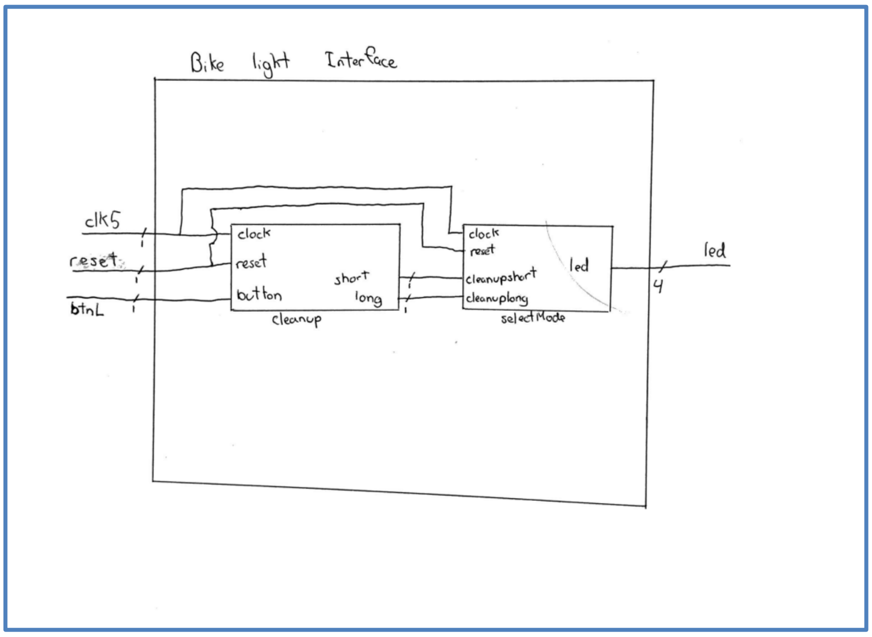
## High-Level Design

### Features

Our bike light has 4 operating modes which can be cycled through using a short press: constant off, constant on, a slow flash for night cycling, and a quick flash for daytime cycling. The bike also has a toggle-able mode that allows the light to be "dimmed" and "brightened", operated through a long press. We wanted short presses to be processed after button release, but long presses to be processed once the threshold for a "long press" was reached. This would allow someone to know when a "long press" was over. The device was also designed to work on "battery insertion" (reset).

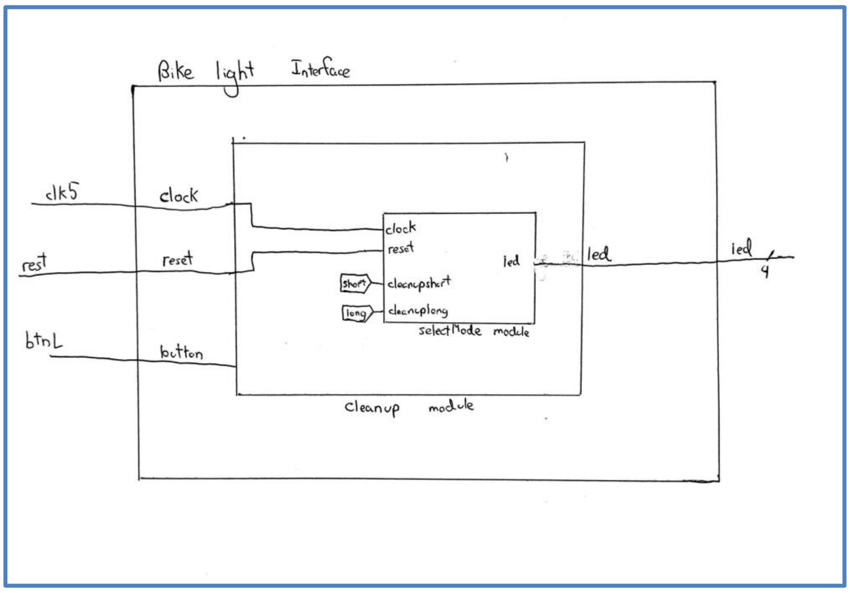
Our design to implement this involves designing two smaller blocks. One block, which will be called 'cleanup', will take the button input, and convert it into usable control signals representing 'short' and 'long' button presses. The other block, called 'selectMode' will take those signals and output a signal that would directly operate the LEDs on the "bikelight" (Nexys-4 display).

Below is our original top-level diagram. As can be seen, the button signal *btnL* is inputted into the module 'cleanup', which will decipher short and long button presses, and eliminate any un-wanted signals due to bounce. These presses, outputted as *long* and *short*, will be inputted into the 'selectMode' module as *cleamupshort* and *cleanuplong* respectively. 'selectMode' will use these control signals to determine what state the led should be in at that moment.



You will notice that the led signal consists of 4 bits, rather than 3. We decided to represent a "brightened" state with 4 consecutive LEDs being active, and a "dimmed" LED with only 2 in sequence. The hardware determining the amount of active LEDs for each state is very simple, and so this combination is very adjustable; We just felt using 4 LEDs was a good way to represent the intended effect.

At the verification stage and the hardware stage, this higher-level diagram was modified slightly. To verify both modules together, we decided to instantiate "selectMode" within "cleanup". This did not change any RTL diagrams, but the output of "cleanup" changed slightly. Instead of outputting *short* and *long* the end product outputted *led*, as shown below.



## User Guide

### Operating Modes

Once batteries are correctly inserted into your bike light, it is 100% ready to function. The light will currently be off; to turn the light on, quickly press and release the large button on top of the bike light. With this, the light will turn and remain on indefinitely. If you want the light to flash slowly, press and release the large button again. If you want the light to flash quickly, press and release the button once more. Finally, if you have finished with your bike light and wish to turn it off, press and release the button once more. From here, the cycle repeats with ‘on’ again.

### Changing the Light Level

If you want, you can change the light level of the bike light. This device has two unique light settings- a dimmed mode, and a full-bright mode. Directly after battery insertion, the light will be in dimmed mode. To switch between the two, press and hold the button until you see the light level increases; when the light level increases, you can release the button. (If the light is currently set to “off”, it might be easier to change to a mode where you can see the change). The current light level will be remembered if you change operating modes, or even if you turn the device off for an extended period. When you come back, the light level will remain the same.

## *Block Design*

After designing the high-level diagrams, we split the work into 2. Jack Coleman designed the “cleanup” module, and Ruairi Hogan designed the “selectMode” module.

As said before, we initially designed each module to be separate blocks that would be instantiated together within the “BikeLightTest.v” file. However, we wanted to verify the combination of both modules together before instantiating them onto the “BikeLightTest.v”. To do this, we decided to instantiate “selectMode” into “cleanup” and made slight modifications to the outputs of each module.

## “cleanup” Module (Jack Coleman)

The “cleanup” module was designed to remove any bounce effects from the button presses and releases, and to detect short and long presses. To do so, I utilized Method 4 as described in the notes: timing how long the button is pressed, and acting if certain thresholds are met.

To do this, I designed a state machine and counter. The signal *timelength* will increment by 1 for every consecutive clock cycle the button was pressed until the long press threshold (600ms, or 3,000,000 clock cycles) was reached, and then remain at that threshold. If the button was ever released, the counter was immediately set to 0. In the meantime, two signals are determined. The wire *en16* turns high only while *timelength* is greater than “clockshort”, a parameter set to 80,000 (or 16ms). This represents when the threshold for a short press has been reached. The wire *en600* turns high when *timelength* is equal to “clocklong”, a parameter set to 3,000,000 (600ms, the long press threshold).

This ensured two details. First, as long as every timing threshold was above the length of time that bounce could happen, bounce would not be able to send false positives on button press or release. Second, stopping the counter when the long press threshold was released ensured that the counter would never overfill to 0 and cause bugs. The counter would effectively stop counting when we stopped caring about the count.

The state machine has 5 states: *idle*, *counting*, *shorthigh*, *longhigh*, and *longpress*. In idle, the state machine is only checking if the button is pressed or not. If the button is pressed, the state machine enters the counting state. Otherwise, the machine remains in idle.

*Counting* is by far the most complex state. While the button is held, the state remains in *counting* unless *en600* is high- in which case the state moves to *longhigh.* This represents the user reaching the long press threshold. If the button is released in this state while *en16* is 0, that means the short press threshold was not released, and was most likely bounce. The State returns to idle. If the button is released while *en16* was reached, a short press has just happened, and the state moves to *shorthigh*.

*Shorthigh* and *longhigh* are very similar with one key difference. The machine remains in each state for exactly one clock cycle, and output a high for either *short* or *long*, respectively. This ensures that the signals representing a short or high press are exactly 1 clock cycle long. The key difference is the next state. The state *shorthigh* will be followed by idle in all cases, as the button is currently un-pressed. In *longhigh,* the button is currently pressed, and so the state moves to *longpress*, where the machine waits until the button is released. This is to allow the action caused by the long press to be processed before the button is released, as we wanted in the design. After the button is released, the state machine returns to *idle*.

### “cleanup” RTL Diagrams

Below is the RTL diagrams for the counter and the state machine. The parameters clockshort and clocklong are defined to allow easy timing changes for verification purposes. I’ve included the RTL diagram before the verification and instantiation changes and after for clarity.

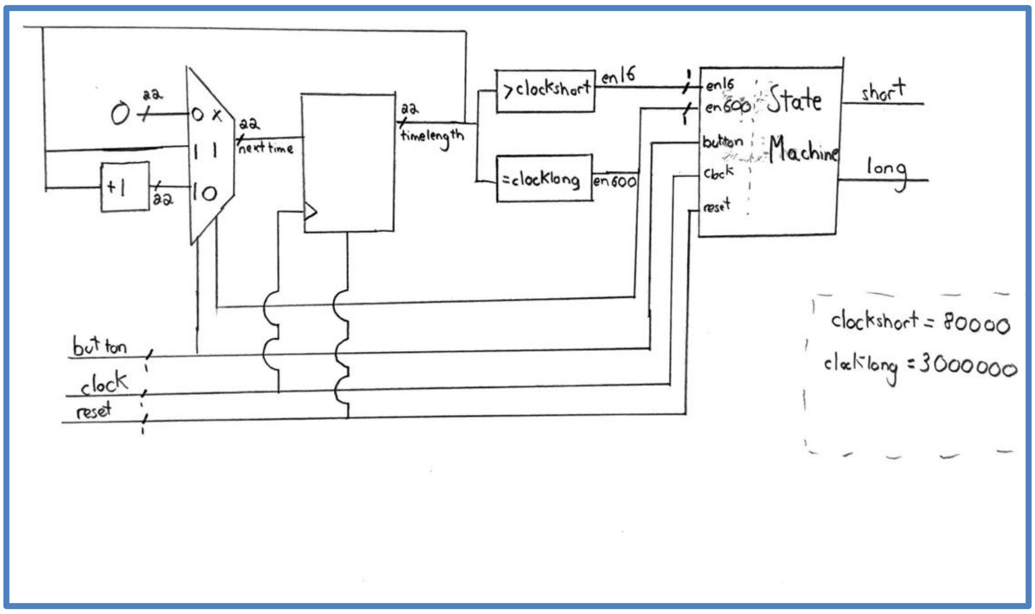


*Figure 1: "cleanup" RTL Diagram Before Verification*

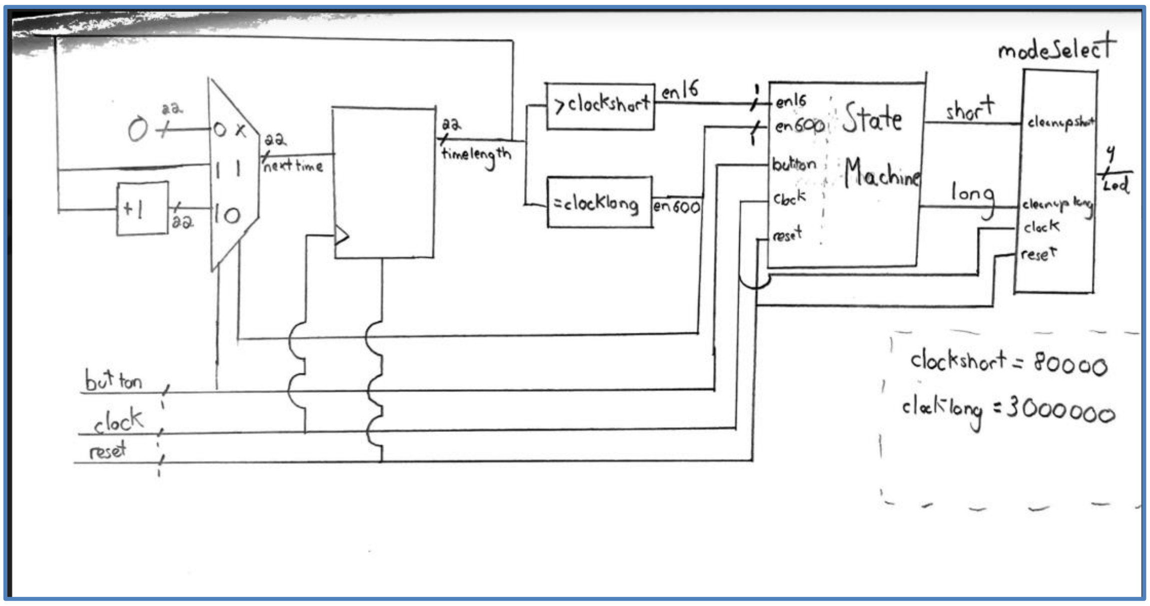


*Figure 2: "cleanup" RTL Diagram After Verification*

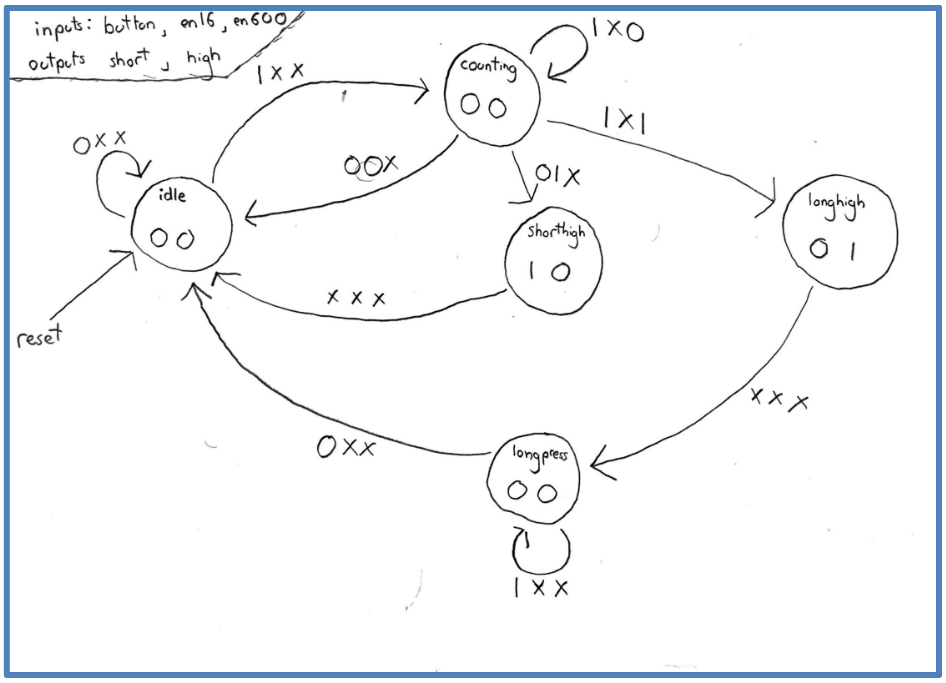


*Figure 3: "cleanup" State Machine*

**"cleanup" Verilog Description**
As said, the Verilog below differs slightly from the RTL diagrams in the output variables.

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////
////////////
// Company: UCD School of Electrical Engineering
// Engineer: Jack Coleman, Ruairi Hogan
//
// Create Date: 10/10/2023
// Project Name: Bike Light Design
// Module Name: cleanup
// Target Devices: Nexys-4 FPGA Board
// Dependencies: modeSelect
// Description: Takes a button input, filters out any presses or
releases due to bounce, and determines long and short presses. Outputs
long and short press signals to modeSelect, which will determine LED
logic.
//
//
//
////////////////////////////////////////////////////////////////////////
////////////


module cleanup(
    input clock, //5 mHz clock
    input reset, //reset button
    input button, //bike light button control
    output [3:0] led); //modification for design
verification/instantiation- simply outputs output of modeSelect


    //Parameters for state machine
    localparam idle=0;
    localparam counting=4;
    localparam longpress=1;
    localparam longhigh=5;
    localparam shorthigh=2;

    //thresholds for long and short presses- commented values are
verification values
    localparam [21:0] clockshort = 22'd80000;//8
    localparam [21:0] clocklong= 22'd3000000;//30

    reg [2:0]state;
    reg [2:0] nextstate;
    wire en16;
    wire en600;
    wire short; //in original state, these are the outputs of module.
Changed in Verification stage to make verifiying/instaniating easier
    wire long;  //in original state, these are the outputs of module.
Changed in Verification stage to make verifiying/instaniating easier

    //  State Register ==============================
    always@(posedge clock)
    if(reset)
    state <=idle;
    else state <= nextstate;

    //  Next State Logic ====================================

    always@(button, en600, en16, state)
    case (state)
    idle: if(button) nextstate=counting; //idle state combinational
logic
          else nextstate=idle;
    counting: if(button && en600) nextstate=longhigh; //counting state
combonational logic
              else if(button) nextstate=counting;
              else if(en16) nextstate=shorthigh;
              else nextstate=idle;
    longpress: if (button) nextstate=longpress; //longpress state
combonational logic
               else nextstate=idle;
    longhigh: nextstate = longpress; //longhigh state combinational
logic
    shorthigh: nextstate = idle; //shorthigh state combinational logic
    default: nextstate = idle;  //default state for errors/bugs
    endcase



    //  Output Logic ==================================
    assign short = (state == shorthigh);
    assign long = (state == longhigh);

  // Clock Logic =====================================
    reg [21:0] timelength;
    reg [21:0] nexttime;


      //clock register
    always @(posedge clock)
    if (reset) timelength <= 1'b0;
    else timelength <= nexttime;

    //multiplexor determining nexttime value
    always @(timelength, button, en600)
    if(!button)
       nexttime = 0;
    else if(en600)
       nexttime = timelength;
    else nexttime = timelength +1'b1;

    //threshold calculations
    assign en16 = (timelength > clockshort);
    assign en600 = (timelength == clocklong);


    //in original forms (based on RTL diagrams and v1 Higher Level
Diagram), short and long would be outputs and below wouldnt exist


    //added this in verification process to test combonation of
modules. Kept in this form for hardware testing
    selectMode xselectMode(
        .cleanupShort(short),
        .cleanupLong(long),
        .clock(clock),
        .reset(reset),
        .led(led)
        );


endmodule
```

## “selectMode” Module (Ruairi Hogan)

**Mode Selection**

The majority of the “selectMode” module is 4 registers acting as counters. The first counter controls the current operating mode. If *cleanupShort* signal is high, the *shortselect* signal increments by one, or returned to 0 if it equals 3.

*Shortselect* is then used as the control signal for the multiplexer that determines the value of the signal *short*. This multiplexer has 4 inputs; it switches between 0 (always off), 1 (always on), 2 (long flash) and 3 (short flash).

The second register is the long counter. This is a counter with an output signal that controls the multiplexer operating the light level. The signal *longSelect* gets inverted from 1 to 0 whenever *cleanupLong* is high.

*Longselect* is the control signal of final multiplexor, determining the value of *led*. The first input is a 4 bit signal {0,*short*,*short*,0} with *short* being the output from the short press multiplexer. This output lights up the second and third LED with the output value of the short press multiplexer. This is the dim mode as only 2 LEDs light up. The second input is a 4-bit signal {*short*, *short*, *short*, *short*}, representing the bright mode. This controls the first four LEDs with the output value of the short press multiplexer. The output from this multiplexer is the output of the entire module and decides which LEDs light up on the FPGA.

**Producing a Flash**

The flash component of this block controls the output of two of the four signals in the *short* multiplexer. The flash is switched between a slow flash and fast flash, based on the last two values of the *shortSelect* control signal.

First there is a multiplexer, controlled by the value of *shortSelect*. This decides if the flash is fast or slow. It switches between two limits for a counter (3’333’333 and 1’250’000), the lower one being for the fast flash and the higher one being for the long flash.

There is a delay counter that increments on each clock cycle. After this counter reaches the limit from the multiplexer above, it makes an enable signal high and sets the counter back to 0.

The enable signal from the delay counter gets fed in as the control signal for a multiplexer. When the enable is high, the multiplexer sets *nextC* of the flash register to be the inverse of *Count*. Otherwise, *nextC=Count*.

*Count* is the output of the flash module. The output signal gets fed into a multiplexer and on each enable high, the output is inverted and inputted back into the register. This means that the value of the output signal is inverted between 1 and 0 every time the delay counter reaches its limit. *Count* represents inputs 2 and 3 of the *short* multiplexor- it is the flash in both cases.

**“selectMode” RTL Diagrams**

Below are the RTL diagrams for the 2 components of the selectMode module. The first is the changing of the modes on the press of the button which takes the inputs cleanupLong, cleanupShort, clock and reset. The second is the flash component, which is the design of what makes the light flash slowly and quickly, which takes the inputs shortSelect, clock and reset.

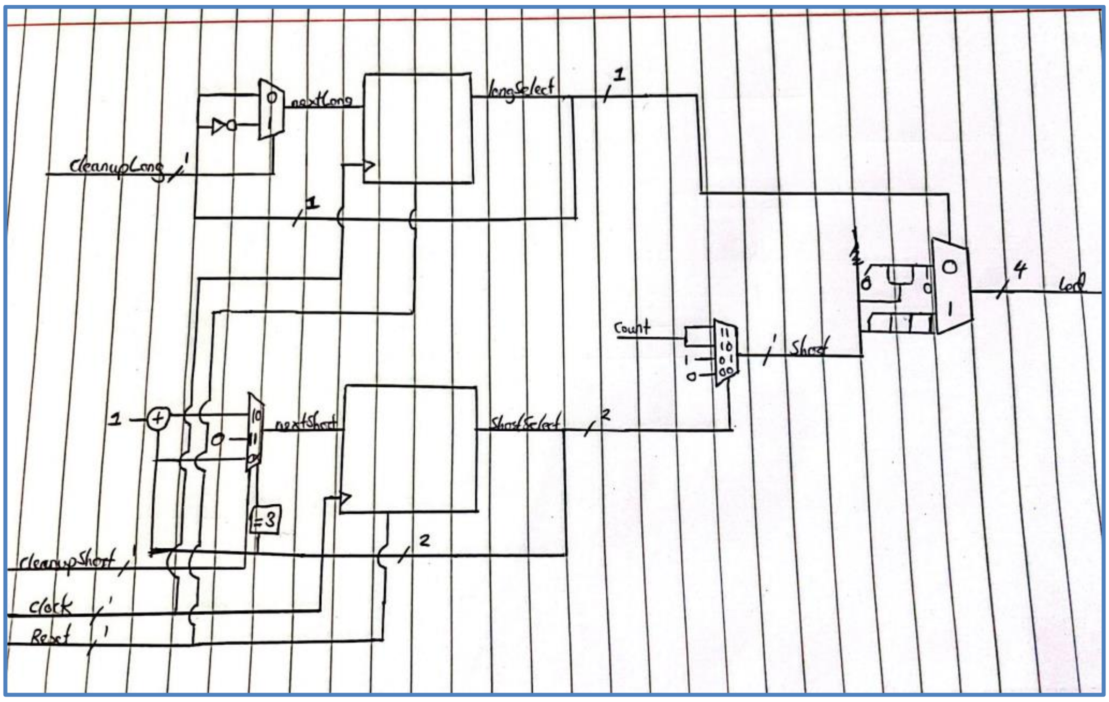


*Figure 4: "selectMode" RTL diagram*

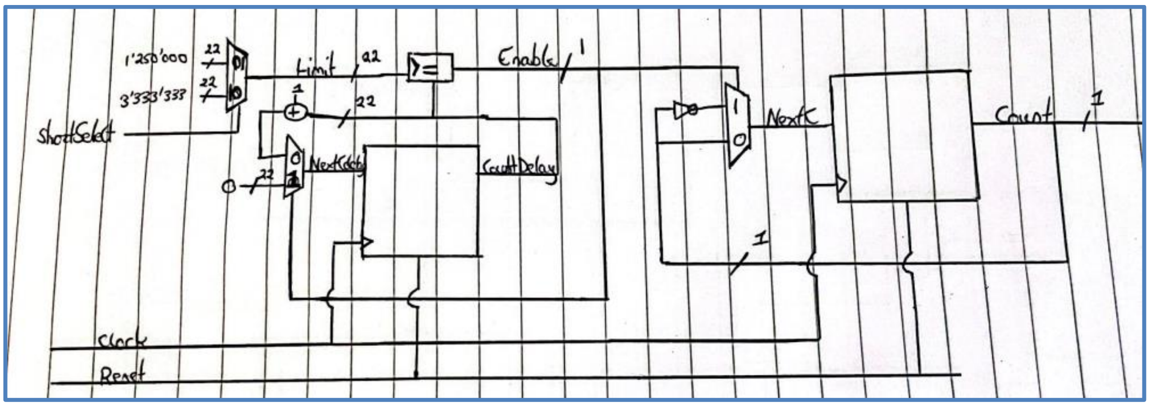


*Figure 5: "selectMode" flash component RTL diagram*

### "selectMode" Verilog Description

```
`timescale 1ns / 1ps


module selectMode(
      input cleanupShort,
      input cleanupLong,
      input clock,
      input reset,
      output reg [3:0] led);

      // Creating Signals for counters
      reg [1:0] shortSelect;
      reg longSelect;
      reg [1:0] nextShort;
      reg nextLong;


    // "Short" counter multiplexer
    always @(shortSelect, cleanupShort)
        if(cleanupShort==1 && shortSelect!=3)//If cleanupShort is 1,
and shortSelect is not 3
            nextShort = shortSelect + 1'b1; // add one to shortSelect
and give to nextShort
        else if(cleanupShort==1 && shortSelect==3) // if cleanup short
is 1 and shortSelect is 3
            nextShort = 1'b0; // nextShort goes back to 0
        else nextShort = shortSelect; //otherwise, continue counting
up 1

    // short counter register
    always @(posedge clock)
        if(reset)
            shortSelect <= 2'd0; //on reset, resets counter to 0
        else shortSelect <= nextShort; // gives the register its
output

    // "Long" counter multiplexer
    always @(longSelect, cleanupLong)
        if(cleanupLong)//If 1, inverts longSelect
            nextLong = ~longSelect;
        else nextLong = longSelect; //otherwise, stays the same
    // long counter register
    always @(posedge clock)
        if(reset)
            longSelect <= 1'd0; //on reset, resets counter to 0
        else longSelect <= nextLong;


//-----------------  Flash -----------------
      // Creating Signals
      reg [21:0] NextCDelay;
      reg [21:0] CountDelay;
      reg Count;
      reg NextC;
      reg Enable;
      reg [21:0] limit;

    // Multiplexer to decide if the Flash is slow or fast
    always @ (shortSelect, limit)
            case(shortSelect)
            2'b10: limit = 22'd3333333;//3333333
            2'b11: limit = 22'd1250000;//1250000
            default: limit = 22'd3333333;//3333333
    endcase

      // determines next value for CountDelay
      always @(CountDelay, limit)
            if(CountDelay >=limit)//If reaches limit, brings back to
0, set enable high
            begin
                  NextCDelay = 22'b0;
                  Enable = 1'd1;
            end
            else
            begin
            NextCDelay = CountDelay+22'b1; //otherwise, continue
Counting up 1 and enable low
            Enable = 1'd0;
            end

      // Count Delay Register
      always @(posedge clock) //synchronous reset
            if(reset)
                  CountDelay <= 22'h0; //on reset, resests countDelay
to 0
            else CountDelay <= NextCDelay; // otherwise it gets
NextCDelay value

      //Flash Register
      always @ (posedge clock)// synchronous reset
            if(reset)//reset enable
                  Count <= 1'h0;
            else Count <= NextC;

//multiplexer for flash
      always @(Count, Enable)
            if(Enable)// inverts Count
                  NextC = ~Count;
            else NextC = Count; // stays the same
      //----------------- Multiplexer for short press select ---------
--------

      reg short;
      always @ (shortSelect, Count)
            case(shortSelect)
            2'b00: short = 1'd0;
            2'b01: short = 1'd1;
            2'b10: short = Count; // flash slow
            2'b11: short = Count; // flash fast
            default: short = 1'd0;
      endcase


      //----------------- Multiplexer for dim/bright-----------------

      reg long;
      always @ (longSelect, short)
            case(longSelect)
            1'd0: led = {1'b0,short,short,1'b0}; // dim
            1'd1: led = {short,short,short,short}; // bright
            default: led = 1'b0;
      endcase


      endmodule
```

## Verification

As can be seen in the headings below, we decided to verify each module separately, and quite thoroughly, before combining the modules into one design, and performing a verification of the entire design.

Further, we decided to independently verify the module that we did not design, to simulate a rough version of black hat verification. It was not a perfect implementation of a black hat verification method, as we talked to each other during the design process and had an idea of how the modules worked, but verifying the design we ourselves did not create definitely allowed for a more conductive mindset during the process, and a more comprehensive analysis.

We went into each test with a list of points we made sure to hit. We found this focussed analysis much more conducive of a productive verification session, than simply saying "I will verify now".

### "cleanup" Module (Ruairi Hogan)

These are some of the more important tests I performed, to ensure that the "cleanup" module works correctly. As said before this module was verified before instantiating "modeSelect.v", so at this point the outputs were still *short* and *long*.

- Button presses less than the short threshold were ignored (8 clock cycles)
- Button presses and releases between the short threshold and the long threshold (30 cycles) produce a 1 clock cycle short pulse.
- Button presses after the long threshold produce a 1 clock cycle pulse at 30 cycles, not when button is released.
- Reset functions properly.

- Holding the button for a long period of time will not produce a second *long* high, or a rogue *small* high if the button is released within a certain window.

This first example a test of the functionality of the reset button, bounce avoidance on both press and release, and short button press detection. The reset button successfully returns *short* and *long* to 0, as well as initialize the timer as not counting. Presses of different clock lengths under 8 cycles were introduced, and neither a *short* high nor a *long* high appeared. When a clock cycle of appropriate length (10 cycles), the *short* signal produced a 1 clock cycle high on button release, which matches the specification.

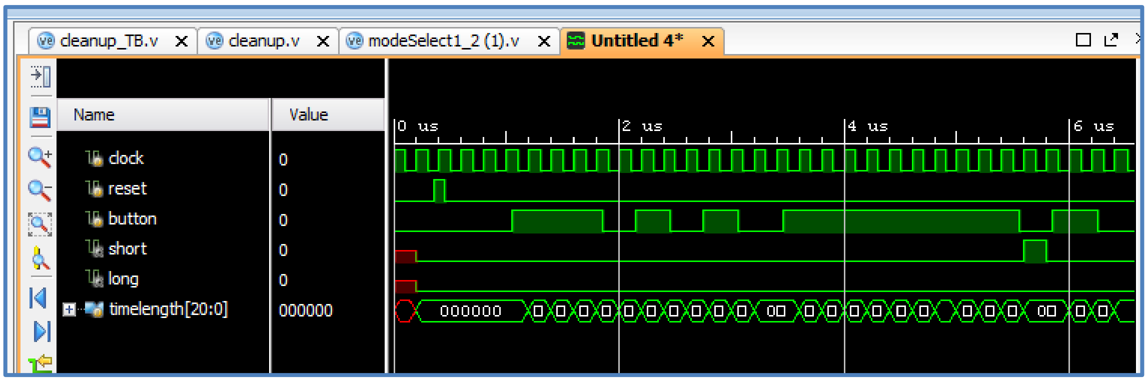


*Figure 6: Cleanup Block Testing*

This is an example of how I tested for long press detection functionality. Again, bounce was applied and ignored on both press and release, and the module successfully outputted a high *long* after 30 clock cycles while the button was still pressed, and not after the button was released.

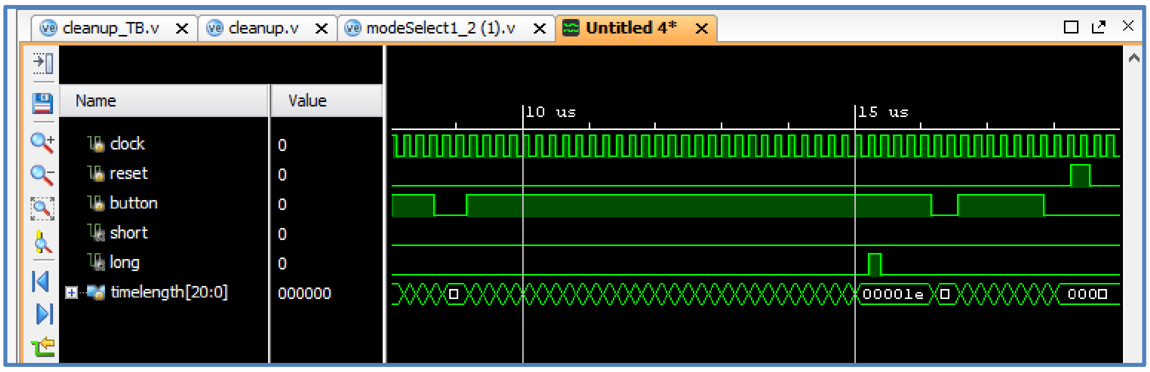


*Figure 7: Cleanup Block Testing*

This method caught the one error before hardware installation. The one error was that one state had a mislabelled next state, meaning that a long press would never be recognized. This was solved before hardware testing.

## "selectMode" Verification (Jack Coleman)

Before verifiying the "selectMode", I made a list of the important cases that I wanted to check. I wanted to ensure that:

- *cleanupShort* and *cleanupLong* advanced the modes to a new mode
- The modes were appearing in the correct order
- All four operating worked in both dim and bright
- Applying reset would bring the light to off and the brightness to 0

- The flashing lights were flashing on the correct frequency

The pictures below demonstrate how I ran these various tests. The first picture displays a verification of the *cleanupShort* input functioning correctly; an active signal advances the operating mode to the next stage. This picture also demonstrates a verification test for the functionality of the modes; We pulsed through each one, assuring that the lights were either all on or off, or flashing at the correct frequency (10 clock cycles and 5 clock cycles respectively). We also ensured that the modes were ordered correctly.

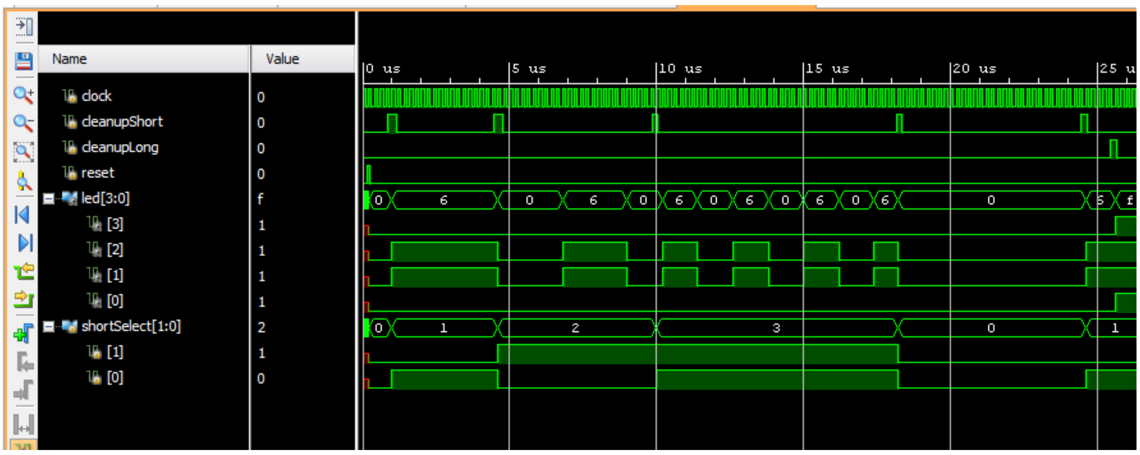


*Figure 8: selectMode block Testing*

The second pictures gives an example of a verification test for the *cleanupLong* signal, the dim/bright modes, as well as the reset button. As can be seen, a pulse from *cleanupLong* correctly changes the LEDs from “dim” to “bright”. The test on the right (40us-50us) shows how the light level setting is remembered through operating mode changes as well. Finally, this shows the correct functioning of the reset button, where the operating mode is returned to 0, and the light level setting is returned to dim.

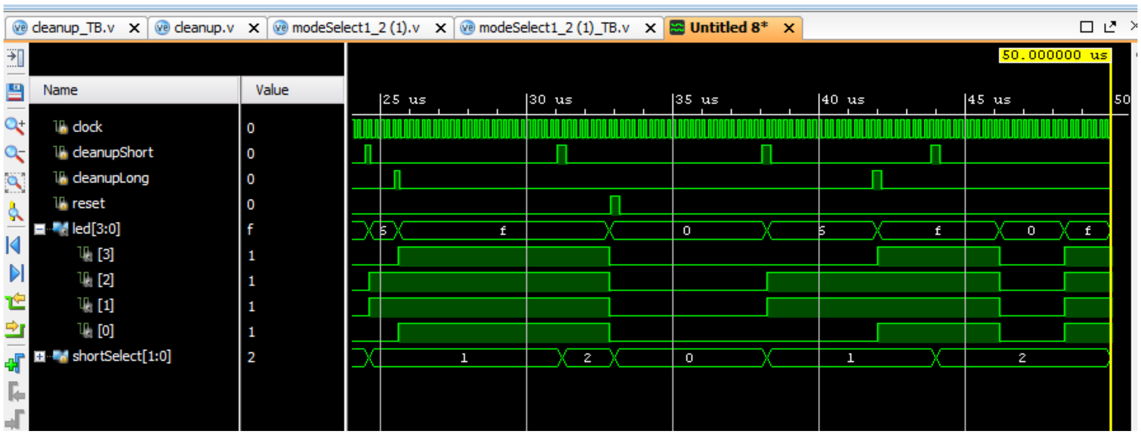


*Figure 9: selectMode block Testing*

This verification plan allowed me to find the one error with the design: the flashing modes were not functioning. With this, we were able to use internal signals to disect the problem: *countDelay* was designed to return to 0 after equalling the value of the *limit* signal. However, the *limit* value could be reduced to change flashing speed, allowing *countDelay* to be above *limit* without reseting to 0. To solve this, we simply changed:

```
if(CountDelay =limit)
```
to
```
if(CountDelay >=limit)
```

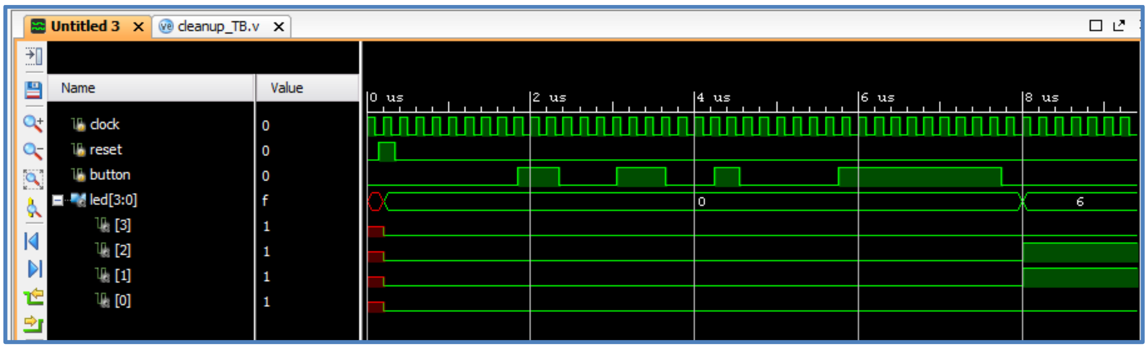
This solution solved the problem.

## Complete Design Verification

After each module was substantially verified, the design as a whole was verified again. At this point, we were already relatively confident in the functionality of the design, but we did this as a “just in case”. Some of the general checks here were simply that of proper instantiation; we knew the modules worked, and we wanted to ensure that signals were properly passed between them. Therefore, we only really were concerned that:

- A simulated short button press changed the operating mode.
- A simulated long button press changed the operating mode.
- LEDs were turning on at all.
- The reset button worked.

This below picture shows a verification test of the design correctly ignoring bounce signals and processing a short button press. It also shows a test of the signal *led* properly being outputted from “modeSelect” to “cleanup”. It also shows the correct functioning of the reset button in giving a value of 0 to the LEDs.



| Name | Value |
|---|---|
| clock | 0 |
| reset | 0 |
| button | 0 |
| led[3:0] | f |
| [3] | 1 |
| [2] | 1 |
| [1] | 1 |
| [0] | 1 |

*Figure 10: Complete design testing*

This final verification test example shows how we verified the functionality of the whole system. We cycle through all 4 modes and both light levels using only the button, and can verifiy mode order, frequency, and output of the whole system, before beginning hardware testing.

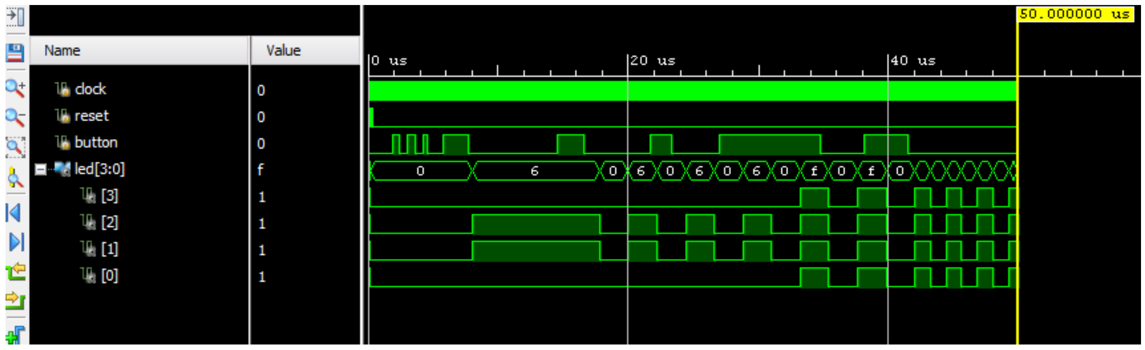


Figure 11: Complete design testing

## Hardware Testing

With verification done, we began hardware testing. Instantiating “modeSelect” into “cleanup” made the hardware process very simple, as we only needed to instantiate one module, as shown below.

```
cleanup xcleanup(
  .clock(clk5),
   .reset(reset),
   .button(btnL),
  .led(led)
   );
```

We recognize that instantiating these modules separately might have been more logical (maybe a bounce module will be needed in the future, and now we will need to modify this to generalize it, rather than already have it finished and verified). However, the simplicity of instantiation onto the hardware was nice.

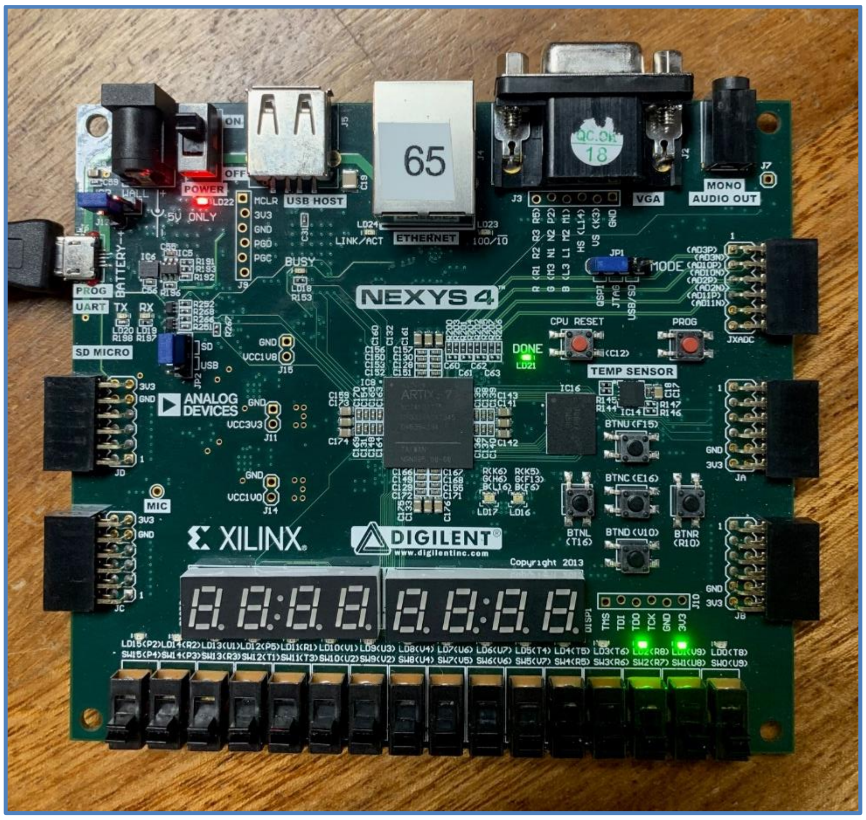
Before programming the Nexys-4 board, we did remember to update the timing values to their correct values. Using local parameters and a single multiplexor made this process very simple and straightforward. We also found our more thorough Verification process made the hardware process much smoother than previous experiences.

Our design worked on the second attempt. See the below code:

```
localparam clockshort = 80000;
localparam clocklong= 3000000;

 [CODE]

reg [20:0] timelength;

 [CODE]

assign en16 = (timelength > clockshort);
assign en600 = (timelength == clocklong);
```

One might notice 3,000,000 cannot fit as a 21-bit binary value, and *timelength* will never reach *clocklong*, and long presses will not work. However, the synthesis tool did not raise any errors or warnings to this, and, as our verification involved lowering *clocklong* to a reasonable level (below 21 bits), our verification would not have caught the error. Even after the hardware test failed, we re-verified the same design to be sure, and the verification looked perfect. For this Verification style to have caught this flaw, our long press verification length would have had to be 2097152 clock cycles long. After quickly solving this, the code worked on the first attempt.

A picture of a bike light is a less than smart way to show off its functionality, but below is a picture of our functioning bike light. This specifically seems to be an example of the light in the "dim" stage.



## Conclusion

After this lab, we felt that we have successfully created a relatively complex bike light. We also feel that we effectively designed the project using RTL diagrams and state machines effectively and transcribed them to Verilog well. Our verification methods were well thought-out, effective, and found the errors in our descriptions.

We think the difference in our product between this project and the last is evident, and we are proud of the result.