Debouncing button

**Concept:**

Clock domain:

A clock domain is a part of a design that has a clock that operates asynchronous to, or has a variable phase relationship with, another clock in the design. For example, a clock and its derived clock (via a clock divider) are in the same clock domain because they have a constant phase relationship. But, 50MHz and 37MHz clocks (whose phase relationship changes over time) define two separate clock domains. Figure 1 illustrates three different clocks in a design, but synchronous to each other. CLK, its inversion and D1 (derived from CLK) are synchronous to each other.

Reference: <https://filebox.ece.vt.edu/~athanas/4514/ledadoc/html/pol_cdc.html>

Asynchronous vs Synchronous Digital Circuits:

From Wikipedia, an **asynchronous circuit**, or **self-timed** circuit, is a sequential digital logic circuit which is not governed by a clock circuit or global clock signal.  This type is contrasted with a **synchronous circuit** in which changes to the signal values in the circuit are triggered by repetitive pulses called a clock signal.

Metastability

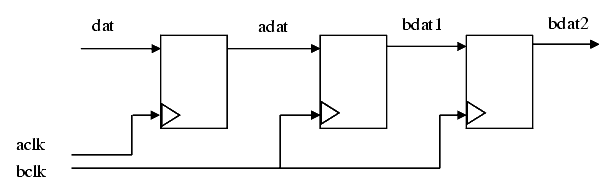
From [Embedded Micro](https://embeddedmicro.com/tutorials/mojo/metastability-and-debouncing), there are setup and hold timing constraints in the flip-flop circuits (mostly is the synchronous circuits). The setup constraint tells you how long before the positive edge of the clock the value on the D side (input) must be stable. The hold time tells you how long after the positive edge of the clock the D side (input) must continue to be stable.

**Q** (output) may take a random value of 0 or 1 or will get stuck somewhere between 0 and 1 if there is setup and hold time violation. This is called **metastability**. This happens when we press a button. It's impossible for us to guarantee that the button won't be pressed and violate the setup and hold constraints. One of the suggested solutions is to have two flip-flops to drastically reduce that chance.  However, this does not solve the metastability problem.

Synchronization

The main responsibility of a synchronizer is to allow sufficient time such that any meta-sable output can settle down to a stable value in the destination clock domain. The most common synchronizer used by designers is two-flip-flop (2-FF) synchronizers as shown below. Usually the control signals in a design are synchronized by 2-FF synchronizers.

In a 2-FF synchronizer, the first flip-flop samples the asynchronous input signal into the destination clock domain and waits for a full destination clock cycle to permit any meta-stability on the stage-1 output signal to decay, then the stage-1 signal is sampled by the same clock into a second stage flip-flop, with the intended goal that the stage-2 signal is now a stable and valid signal synchronized into the destination clock domain. It is theoretically possible for the stage-1 signal to still be sufficiently meta-stable by the time the signal is clocked into the second stage to cause the stage-2 signal to also go meta-stable.



<https://filebox.ece.vt.edu/~athanas/4514/ledadoc/html/pol_cdc.html>

Debonucing

When you press a button, there is a chance that the button will not simply go from open to close. Since a button is a mechanical device, the contacts can bounce. (From [Embedded Micro](https://embeddedmicro.com/tutorials/mojo/metastability-and-debouncing)) For a short period after the button is pressed the value you read from an IO pin may toggle between 0 and 1 a few times before settling on the actual value.

To debounce a button, you just need to require that for a button to register as being pressed, it must look like it's being pressed for a set amount of time. In this case, being pressed is when the value of the button is 1. If you read enough 1's in a row it is safe to assume that the button has stopped bouncing and you can register one button press. If you fail to do this and you are using the button to increment a counter, then the counter may increase by more than 1 per button press since it will appear that each bounce was a separate press.

**Design:**

There are following inputs and outputs

|  |  |  |
| --- | --- | --- |
| Inputs / Outputs | Signal name | Description |
| Input | clk | Clock signal to synchronize the button input |
| Input | btn | Button pressed |
| Output | LED | Light up LED when button is pressed |
| Output | dbsig | Output the debounced signal to the scope |
| Output | button\_out1 | Output the flip flop 1 signal to scope |
| Output | button\_out2 | Output the flip flop 2 signal to scope |

There are following internal signals

|  |  |  |
| --- | --- | --- |
| Type of signal | Signal name | Description |
| Register (reg) | button\_ff1 | Btn flip-flop1 for synchronization |
| Register (reg) | button\_ff2 | Btn filp-flop2 for synchronization |
| Register (reg) | [20:0]count | 21 bits counter to increment when button is pressed or released |
| Constant (Parameter) | threshold | Gauge how long the button should be pressed. In this case, we want 2ms, so the number we need to count is 100,000 for 100MHz clock |

How does the logic work?

Firstly, we use two flip-flops (reg button\_ff1 & reg button\_ff2 to synchronize the button signal to the clock domain “clk”. When the push-button is pushed or released, we increment or decrement the counter “count”. The counter has to reach threshold before we decide that the push-button state has changed. We implement this by conditional statements:

**Condition 1:** Button flip-flop 2 (reg button\_ff2) is high “1” and count ***“***count” isn't at the count limit.

This makes sure that we won't count up at the limit. If this condition matches, we will count up (this means btn pressed)

**Condition 2:** Button flip-flop 2 (reg button\_ff2) is low “0” and the count has at least 1 in it.

This makes sure no subtraction when count is 0. If this condition matches, we will count down (btn released)

Now, we need to decide when the button is pressed or not. To do this, we create a constant (parameter) “threshold” gauge how long button pressed to determine it is in pressed state. If the count is greater the threshold, the debounced signal “dbsign” will be high “1”. Otherwise, it will be low “0”.

We first simulate the logic and then scope out two signals (button\_out1 and button\_out2) through Pmod connectors to the Logic Analyzer and look at the timing diagram. In addition, we assign LED to debounced signal (dbsig), i.e. if the dbsig is “1”, LED lights up. Otherwise, it will be off.

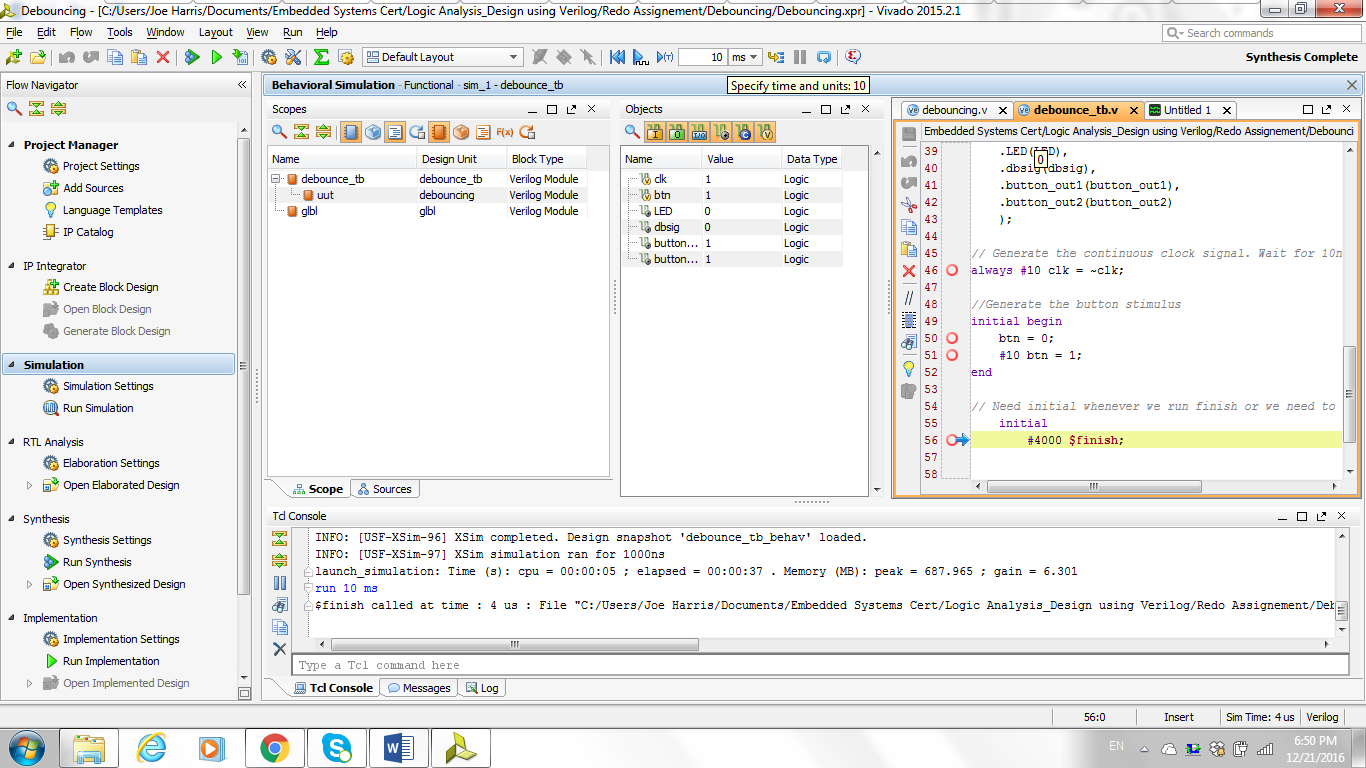
**Simulation:**

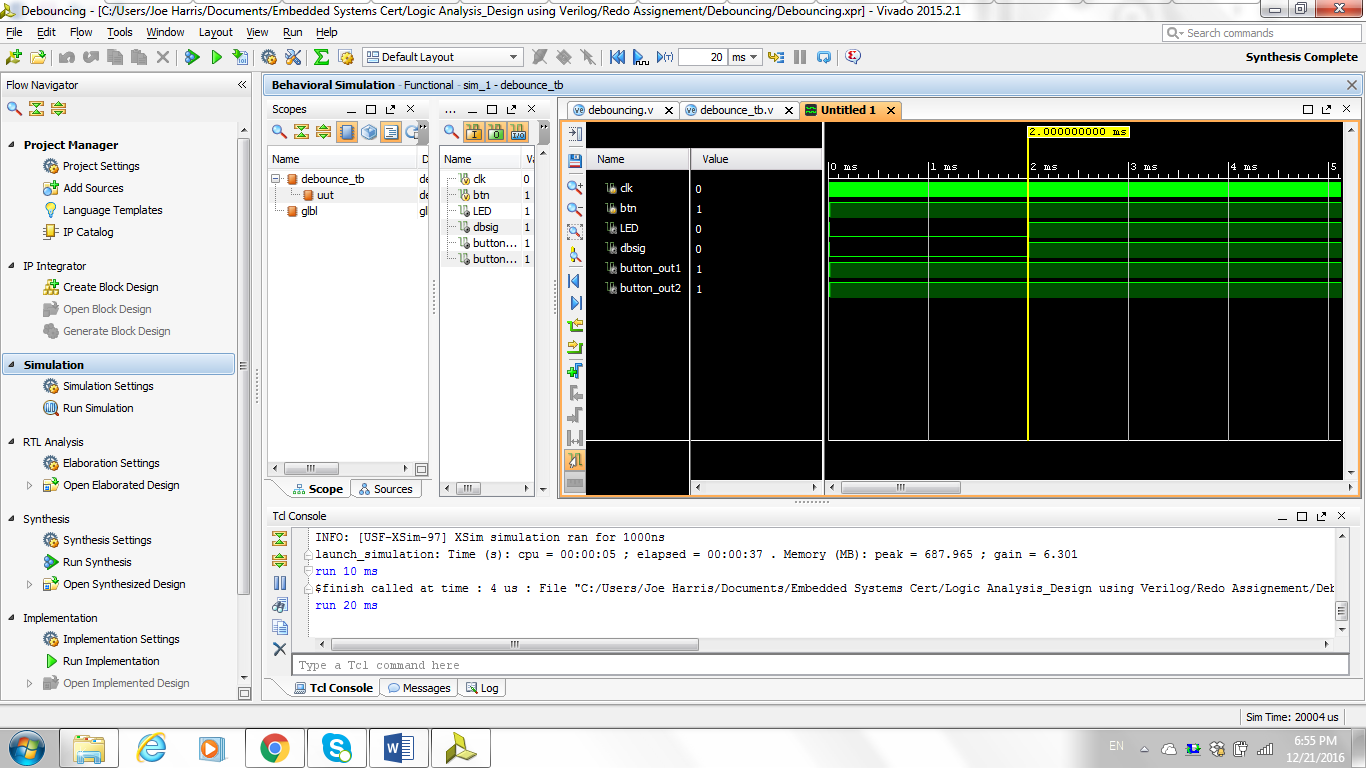
Follow the step to create the testbench and run the simulation <http://www.instructables.com/id/How-to-Use-Vivado-Simluation/>

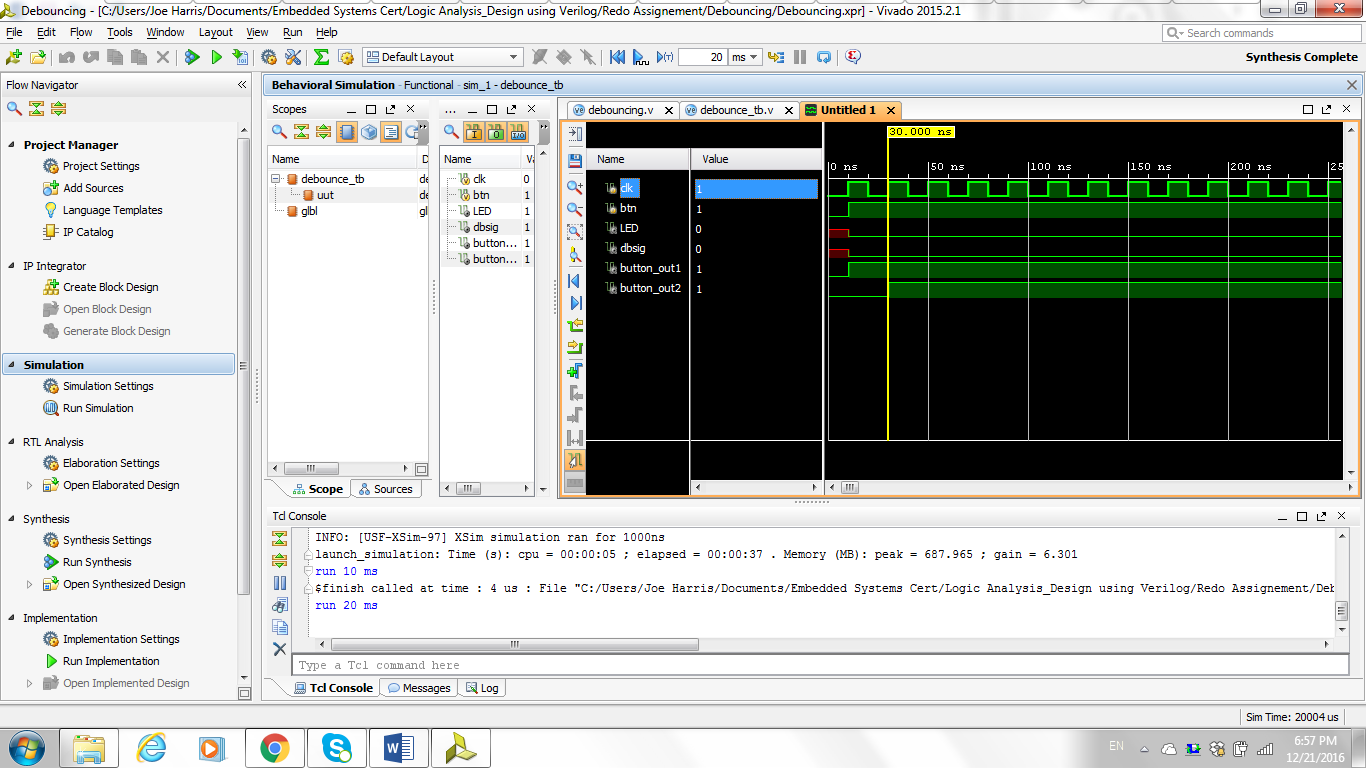
A 20ns clock is created and btn input is set as 0 and then 1 after 10ns. The dbsig and LED signal will be “1” at 2ms. Run the simulation at 20ms by change the simulation time (see below picture)

Remember to create two procedure block (one always for clock generation and one for stimulus (btn)) to avoid any intra signal delay.

There is 30ns delay between button\_ff1 and button\_ff2. It shows in the simulation of button\_out1 and button\_out2.







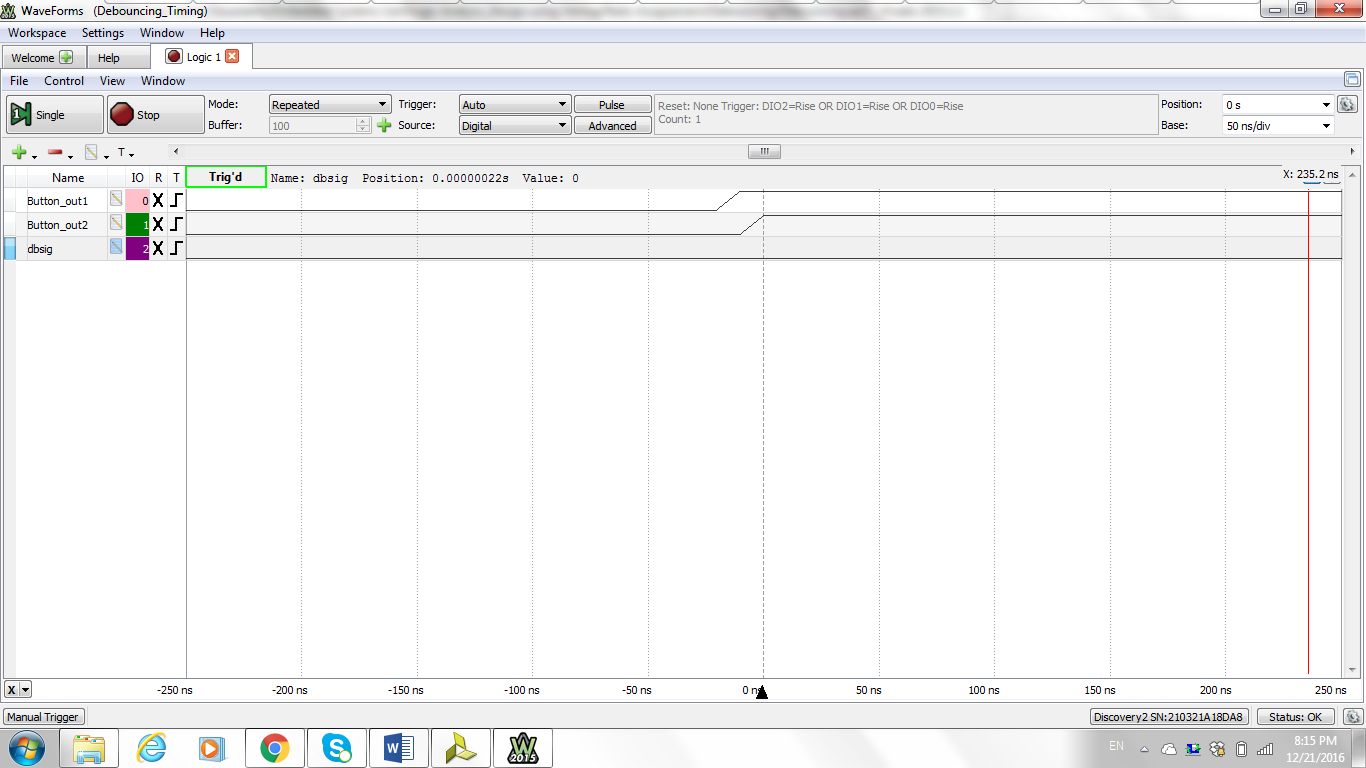
**Implementation:**

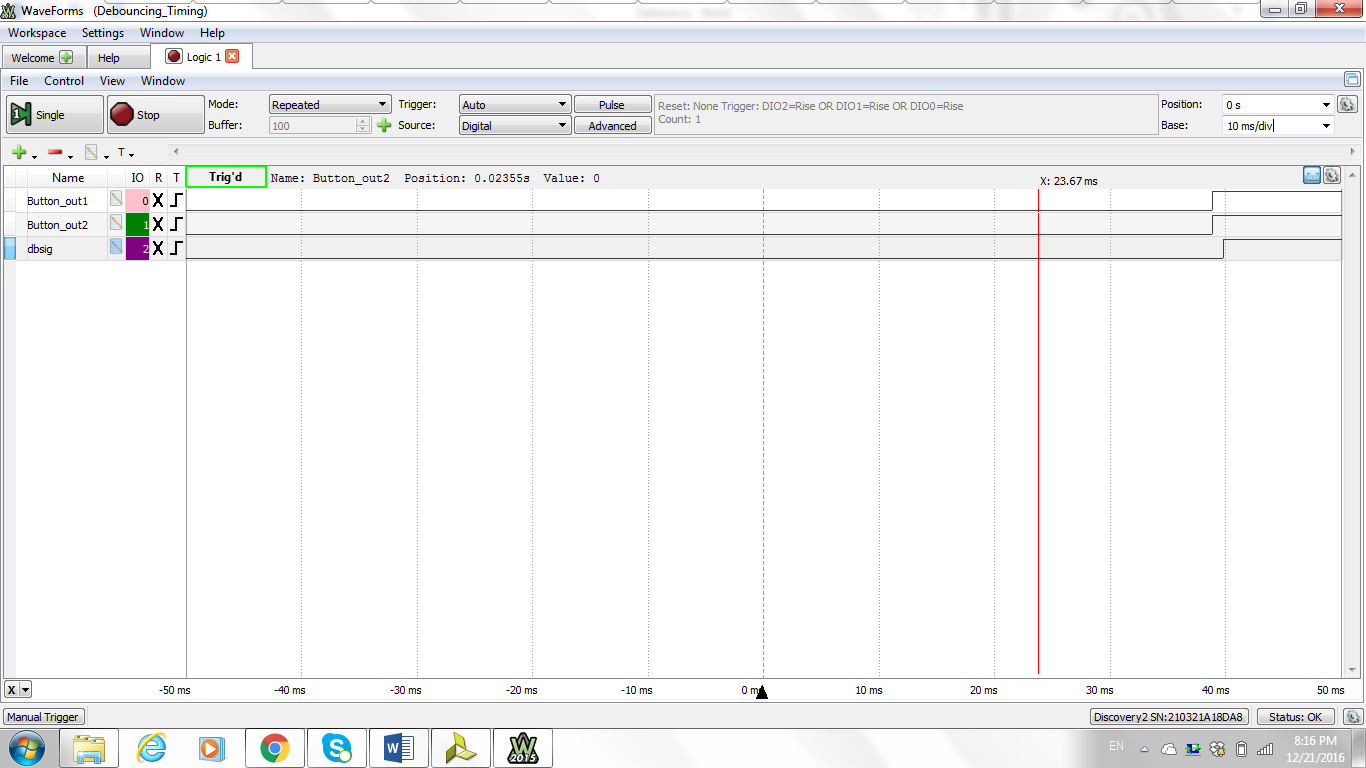
Create constraint file from Vivado. Refer to <http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug935-vivado-io-clock-planning-tutorial.pdf> (see Lab 2: Post-Synthesis I/O Planning) and <http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug945-vivado-using-constraints-tutorial.pdf> for timing constraint

Then, run the synthesis and implementation and eventually generate bitstream file to program it on FPGA. (If you are not familiar with these two processes, you can take a look at <http://www.instructables.com/id/How-to-use-Verilog-and-Basys-3-to-do-3-bit-binary-/> from step 8-11.

**Timing Analysis through Logic Analyzer**

We port button flip flop signals (button\_out1 and button\_out2) and dbsig to Pmod port JC1, JC2 and JC3 and then connect ports to Digilent Analog Discovery 2 digital channels 0, 1 and 2





**Verilog Code:**

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 07/17/2016 05:01:56 PM

// Design Name:

// Module Name: debouncing

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module debouncing #(parameter threshold = 100000 )// set parameter thresehold to guage how long button pressed

(

input clk, //clock signal

input btn, //input button

output LED, //LED output. Light up when btn is pressed

output reg dbsig, //debounced signal to logic analyzer

output button\_out1, //flip-flop1 signal to logic analyzer

output button\_out2 //flip-flop2 signal to logic analyzer

);

reg button\_ff1 = 0; //button flip-flop for synchronization. Initialize it to 0

reg button\_ff2 = 0; //button flip-flop for synchronization. Initialize it to 0

reg [20:0]count = 0; //20 bits count for increment & decrement when button is pressed or released. Initialize it to 0

// First use two flip-flops to synchronize the button signal the "clk" clock domain

always @(posedge clk)begin

button\_ff1 <= btn;

button\_ff2 <= button\_ff1;

end

// When the push-button is pushed or released, we increment or decrement the counter

// The counter has to reach threshold before we decide that the push-button state has changed

always @(posedge clk) begin

if (button\_ff2) //if button\_ff2 is 1

begin

if (~&count)//if it isn't at the count limit. Make sure won't count up at the limit. First AND all count and then not the AND

count <= count+1; // when btn pressed, count up

end else begin

if (|count)//if count has at least 1 in it. Make sure no subtraction when count is 0

count <= count-1; //when btn relesed, count down

end

if (count > threshold)//if the count is greater the threshold

dbsig <= 1; //debounced signal is 1

else

dbsig <= 0; //debounced signal is 0

end

assign LED = dbsig; // assign debounced signal to LED

assign button\_out1 = button\_ff1; //assign button\_out1 to button\_ff1

assign button\_out2 = button\_ff2; //assign button\_out2 to button\_ff2

endmodule

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 12/19/2016 09:21:44 PM

// Design Name:

// Module Name: debounce\_tb

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module debounce\_tb();

//Inputs in the module enable\_sr. Need to use register type

reg clk = 0; //clock signal

reg btn; //input button

//Outputs in the module enable\_sr. Need to use net type

wire LED; //LED output. Light up when btn is pressed

wire dbsig; //debounced signal to logic analyzer

wire button\_out1; //flip-flop1 signal to logic analyzer

wire button\_out2; //flip-flop2 signal to logic analyzer

// Instantiate the Unit Under Test (UUT) for module debouncing

debouncing uut(

.clk(clk),

.btn(btn),

.LED(LED),

.dbsig(dbsig),

.button\_out1(button\_out1),

.button\_out2(button\_out2)

);

// Generate the continuous clock signal. Wait for 10ns. Period is 20ns

always #10 clk = ~clk;

//Generate the button stimulus

initial begin

btn = 0;

#10 btn = 1;

end

// Need initial whenever we run finish or we need to include it in the begin end

initial

#4000 $finish;

endmodule