Go Board Project 4 – Debounce A Switch

Video: https://youtu.be/plGDQDyLDR0

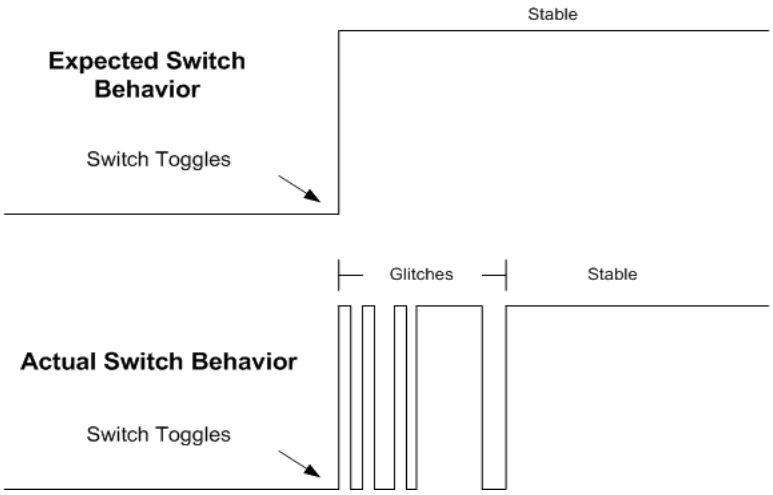
Link: https://nandland.com/project-4-debounce-a-switch/

And learn how time works inside of an FPGA!

The previous project introduced Flip-Flops, and we made an LED toggle when we pushed a button. However, on the last project there was a problem: **pushing the Switch did not consistently toggle the state of the LED**.
This is caused by the switches bouncing. If you are ever using physical switches, you should be careful. Physical switches such as push buttons and toggle switches are all subject to bouncing. Bouncing occurs when the switch is toggled or flipped.
It happens in all switches as a result of the metal contacts coming together and apart quickly before they have time to settle out.

I also created a YouTube video for this project, should you prefer to follow along with that.

The image below shows the transition from logic low to logic high of a switch closing. One would expect a nice clean transition. However, the spikes are where the switch contacts create glitches on the line. This is what was happening on the previous project. If your eye was fast enough to detect it, you would see the LED turning off and on several times very rapidly while the switch was bouncing around.



Glitches on Switch

The code in the project was looking for a falling edge, but it was seeing many falling edges.
If it saw an odd number of falling edges during the bouncing of the switch, then the LED toggled successfully. If it saw an even number of falling edges, the LED didn't appear to change state to the observer.
The number of bounces on the switch is somewhat random, so pushing the switch enough times got the LED to toggle successfully. But we need to fix this.
This switch is in need of some debounce filtering!

Debounce Filter, Dealing With Time

The way I chose to create the filter was to make sure that the switch input is stable for some amount of time and only allow the output to change if the input is stable.
So we need to know how much time has passed since the switch has been stable.

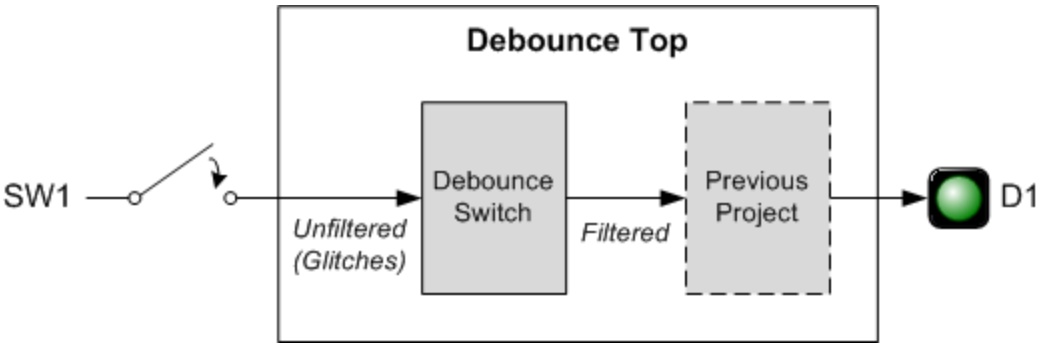
But wait, time does not exist inherently in an FPGA! This is a fundamental concept that a lot of people miss when first starting in FPGA design. While there are parts of the languages that refer to time, these parts of the languages are *not synthesizable*.

We already talked about Synthesis. Synthesis is part of your build process where the FPGA tool turns your VHDL or Verilog into Flip-Flops and LUTs and other components.
Synthesis cannot synthesize anything relating to time. It's just not possible! There's actually a decent number of keywords in both languages that are simply ignored or create errors for the synthesis tool. This means they are not synthesizable.

Okay, what the hell is going on? I just told you that you can't synthesize time. But I told you previously that we are looking for the Switch to be stable for some amount of time. Hey nandland guy (my name is Russell) you are crazy! Maybe so, but regardless we need to be a bit clever about how we deal with time in an FPGA.
The way to know how much time has passed in an FPGA is to count clock cycles.

From the previous project we learned that the Go Board has a clock that oscillates at 25 MHz.
25 MHz means that each clock cycle takes 40 nanoseconds (ns). So… how many clock cycles would it take for 400 ns? Answer: 10. And 4000 ns? Answer: 100. See where I'm going with this? If you want to keep track of time in your FPGA, you need to count clock cycles, while knowing what the period of your particular clock is. This is going to be critical in our debounce project. Let's get to it.

Project Description



This project should add a debounce filter to the code from the previous project to ensure that a single press of the button toggles the LED

You are welcome to try this project on your own first and see what you come up with. My solution for both VHDL and Verilog are below.

You might be tempted to use a **for loop** to implement a counter. Let me settle this right now, **for loops in hardware languages do not work the way that you might expect them to from a language like C or Java**.
For loops are an advanced concept, so for now do not use them. **Instead, use a counter!**

Both pieces of code behave exactly the same, so I'll explain how they both work.
There is a signal called **r_State** which stores the filtered state of the switch input. The logic that drives **r_State** filters out the fast transitions on **i_Switch** and only allows **r_State** to change when the switch has been stable for a long enough amount of time.
In this case, I chose **10 milliseconds (ms).**

There is one main process (in the VHDL code) and always block (in the Verilog code) that is continuously looking for a change on the **i_Switch** input.
If i_Switch is different from r_State (filtered output), then a counter is incremented.
Once the counter reaches its limit, r_State *samples* the value from i_Switch.
The word samples means that it looks at it for one or more clock cycles, implying sequential or registered logic.

This project is the first one in which we will be instantiating a block of code from within another module. In both cases below, we create the Debounce Filter logic, but this is not done at the highest level of our design.
At the highest level of our design (the main FPGA interface) we **instantiate** the Debounce filter.
Instantiation means that create an instance of that code.
Instantiation is used all the time in more complicated designs, as it promotes reusability and portability.

Now, in any future project that we do that requires debouncing, we can simply instantiate the Debounce_Switch module.
Let's take a look at the code.

VHDL Code

The VHDL code below introduces a few new concepts. The first thing you might notice is that there are **two files**.
Debounce_Project_Top is the top level of the FPGA build, which goes to the physical pins on the Go Board.
Debounce_Switch is a lower level module which gets instantiated by the top level architecture. You can see how that's done below.
Note that the **=>** is used to map the inner signals on the left to the outer signals on the right.

The code in the top level should be *very* close to the code from the previous project where we toggled the LED.
All we are adding is the ability to debounce the input switch, so rather than **i_Switch_1** being used in the process, we need to use the output of the **Debounce_Switch** module.
We need to create an intermediary signal **(w_Switch_1),** which will serve to wire up the output of **Debounce_Switch** to be able to be used in **Debounce_Project_Top**.

For any signals that are wires, I use the prefix **w_.** Wires mean that the signal will not be turned into a register, it just is used to create a wire between two points.
Putting a prefix on the signal is not required, but it helps me keep my code organized and clear.

VHDL Code – Debounce_Project_Top.vhd:

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity Debounce_Project_Top is
  port (
    i_Clk       : in  std_logic;
    i_Switch_1  : in  std_logic;
    o_LED_1     : out std_logic
    );
end entity Debounce_Project_Top;

architecture RTL of Debounce_Project_Top is

  signal r_LED_1    : std_logic := '0';
  signal r_Switch_1 : std_logic := '0';
  signal w_Switch_1 : std_logic;

begin

  -- Instantiate Debounce Filter
  Debounce_Inst : entity work.Debounce_Switch
    port map (
      i_Clk    => i_Clk,
      i_Switch => i_Switch_1,
      o_Switch => w_Switch_1);

  -- Purpose: Toggle LED output when w_Switch_1 is released.
  p_Register : process (i_Clk) is
  begin
    if rising_edge(i_Clk) then
      r_Switch_1 <= w_Switch_1;         -- Creates a Register

      -- This conditional expression looks for a falling edge on i_Switch_1.
      -- Here, the current value (i_Switch_1) is low, but the previous value
      -- (r_Switch_1) is high.  This means that we found a falling edge.
      if w_Switch_1 = '0' and r_Switch_1 = '1' then
        r_LED_1 <= not r_LED_1;         -- Toggle LED output
      end if;
    end if;
  end process p_Register;

  o_LED_1 <= r_LED_1;

end architecture RTL;
```

VHDL Code – Debounce_Project_Top.vhd:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity Debounce_Project_Top is
5   port (
6     i_Clk       : in  std_logic;
7     i_Switch_1  : in  std_logic;
8     o_LED_1     : out std_logic
9     );
10end entity Debounce_Project_Top;
11
12architecture RTL of Debounce_Project_Top is
13
14  signal r_LED_1    : std_logic := '0';
15  signal r_Switch_1 : std_logic := '0';
16  signal w_Switch_1 : std_logic;
17
18begin
19
20  -- Instantiate Debounce Filter
21  Debounce_Inst : entity work.Debounce_Switch
22    port map (
23      i_Clk    => i_Clk,
24      i_Switch => i_Switch_1,
25      o_Switch => w_Switch_1);
26
27  -- Purpose: Toggle LED output when w_Switch_1 is released.
28  p_Register : process (i_Clk) is
29  begin
30    if rising_edge(i_Clk) then
```

```
31      r_Switch_1 <= w_Switch_1;         -- Creates a Register
32
33      -- This conditional expression looks for a falling edge on i_Switch_1.
34      -- Here, the current value (i_Switch_1) is low, but the previous value
35      -- (r_Switch_1) is high.  This means that we found a falling edge.
36      if w_Switch_1 = '0' and r_Switch_1 = '1' then
37        r_LED_1 <= not r_LED_1;         -- Toggle LED output
38      end if;
39    end if;
40  end process p_Register;
41
42  o_LED_1 <= r_LED_1;
43
44end architecture RTL;
```

Now let's look at the actual debounce filter logic. There's a new package that we are using: **numeric_std**.
Numeric_Std is a package that allows us to perform addition. Without a reference to this package, the **+** operator below will not work correctly. Note that there's a different package called **std_logic_arith** that is often included in VHDL designs. You should never use std_logic_arith because it is not an official ieee supported package.

The debounce filter introduces **constants**.
Constants are set once and cannot be overridden. They allow us to have a nice clean piece of code. Now if the **c_DEBOUNCE_LIMIT** changes, we only need to change the constant initialization value.

We also introduce a new type: **integer**.
When using integers in VHDL, it's recommended to put a range on it.
This way, the synthesis tool knows exactly the range of the integer and will only synthesize the number of bits that are actually necessary.
Additionally, when you run a simulation, the simulator always checks that the integer remains within bounds. If it ever goes higher or lower than you allow, the simulator will give you an error, which is usually a quick way to find that you're doing something wrong.

VHDL Code – Debounce_Switch.vhd:

```
1----------------------------------------------------------------------------
2-- File downloaded from http://www.nandland.com
3----------------------------------------------------------------------------
4-- This module is used to debounce any switch or button coming into the FPGA.
```

```vhdl
-- Does not allow the output of the switch to change unless the switch is
-- steady long enough time.
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Debounce_Switch is
  port (
    i_Clk    : in  std_logic;
    i_Switch : in  std_logic;
    o_Switch : out std_logic
    );
end entity Debounce_Switch;

architecture RTL of Debounce_Switch is

  -- Set for 250,000 clock ticks of 25 MHz clock (10 ms)
  constant c_DEBOUNCE_LIMIT : integer := 250000;

  signal r_Count : integer range 0 to c_DEBOUNCE_LIMIT := 0;
  signal r_State : std_logic := '0';

begin

  p_Debounce : process (i_Clk) is
  begin
    if rising_edge(i_Clk) then

      -- Switch input is different than internal switch value, so an input is
      -- changing.  Increase counter until it is stable for c_DEBOUNCE_LIMIT.
      if (i_Switch /= r_State and r_Count < c_DEBOUNCE_LIMIT) then
        r_Count <= r_Count + 1;

      -- End of counter reached, switch is stable, register it, reset counter
      elsif r_Count = c_DEBOUNCE_LIMIT then
        r_State <= i_Switch;
        r_Count <= 0;

      -- Switches are the same state, reset the counter
      else
        r_Count <= 0;

      end if;
    end if;
  end process p_Debounce;

  -- Assign internal register to output (debounced!)
  o_Switch <= r_State;

end architecture RTL;
```

VHDL Code – Debounce_Switch.vhd:

```
1 ----------------------------------------------------------------------------
2 -- File downloaded from http://www.nandland.com
3 ----------------------------------------------------------------------------
4 -- This module is used to debounce any switch or button coming into the FPGA.
5 -- Does not allow the output of the switch to change unless the switch is
6 -- steady long enough time.
7 ----------------------------------------------------------------------------
8 library ieee;
9 use ieee.std_logic_1164.all;
10use ieee.numeric_std.all;
11
12entity Debounce_Switch is
13  port (
14    i_Clk    : in  std_logic;
15    i_Switch : in  std_logic;
16    o_Switch : out std_logic
17    );
18end entity Debounce_Switch;
19
20architecture RTL of Debounce_Switch is
21
22  -- Set for 250,000 clock ticks of 25 MHz clock (10 ms)
23  constant c_DEBOUNCE_LIMIT : integer := 250000;
24
25  signal r_Count : integer range 0 to c_DEBOUNCE_LIMIT := 0;
26  signal r_State : std_logic := '0';
27
28begin
```

```
29
30  p_Debounce : process (i_Clk) is
31  begin
32    if rising_edge(i_Clk) then
33
34      -- Switch input is different than internal switch value, so an input is
35      -- changing.  Increase counter until it is stable for c_DEBOUNCE_LIMIT.
36      if (i_Switch /= r_State and r_Count < c_DEBOUNCE_LIMIT) then
37        r_Count <= r_Count + 1;
38
39      -- End of counter reached, switch is stable, register it, reset counter
40      elsif r_Count = c_DEBOUNCE_LIMIT then
41        r_State <= i_Switch;
42        r_Count <= 0;
43
44      -- Switches are the same state, reset the counter
45      else
46        r_Count <= 0;
47
48      end if;
49    end if;
50  end process p_Debounce;
51
52  -- Assign internal register to output (debounced!)
53  o_Switch <= r_State;
54
55end architecture RTL;
```

Verilog Code

The Verilog code below introduces a few new concepts.
The first thing you might notice is that there are two files. **Debounce_Project_Top** is the top level of the FPGA build, which goes to the physical pins on the Go Board.
Debounce_Switch is a lower level module which gets instantiated by the top level module. You can see how that's done below. Note that the signals in the parenthesis are the ones in the current file. The files before the parenthesis are the ones in the instantiated module.

The code in the top level should be *very* close to the code from the previous project where we toggled the LED. All we are adding is the ability to debounce the input switch, so rather than **i_Switch_1** being used in the process, we need to use the output of the **Debounce_Switch** module. We need to create an intermediary signal **(w_Switch_1),** which will serve to wire up the output of **Debounce_Switch** to be able to be used in **Debounce_Project_Top**.

For any signals that are wires, I use the prefix w_. Wires mean that the signal will not be turned into a register, it just is used to create a wire between two points. Putting a prefix on the signal is not required, but it helps me keep my code organized and clear.

Verilog Code – Debounce_Project_Top.v:

```
1module Debounce_Project_Top
2  (input  i_Clk,
3   input  i_Switch_1,
4   output o_LED_1);
5
6  reg  r_LED_1    = 1'b0;
7  reg  r_Switch_1 = 1'b0;
8  wire w_Switch_1;
9
10 // Instantiate Debounce Module
11 Debounce_Switch Debounce_Inst
12 (.i_Clk(i_Clk),
13  .i_Switch(i_Switch_1),
14  .o_Switch(w_Switch_1));
15
16 // Purpose: Toggle LED output when w_Switch_1 is released.
17 always @(posedge i_Clk)
18 begin
19   r_Switch_1 <= w_Switch_1;         // Creates a Register
20
21   // This conditional expression looks for a falling edge on w_Switch_1.
22   // Here, the current value (i_Switch_1) is low, but the previous value
23   // (r_Switch_1) is high.  This means that we found a falling edge.
24   if (w_Switch_1 == 1'b0 && r_Switch_1 == 1'b1)
25   begin
26     r_LED_1 <= ~r_LED_1;         // Toggle LED output
```

```
27    end
28  end
29
30  assign o_LED_1 = r_LED_1;
31
32endmodule
```

Now let's look at the actual debounce filter logic.

The debounce filter introduces **parameters**.
Parameters are local constants. They allow us to have a nice clean piece of code.
Now if the **c_DEBOUNCE_LIMIT** changes, we only need to change the parameter initialization value.

We also introduce a subtle addition after the **reg** keyword.
In this case, I declared **r_Count** to be an **18-bit vector [17:0].**
This is read verbally as "seventeen down to zero".
The reason that I chose to use 18-bits to store the **r_Count** value is that this is the minimum number of bits required to store the value **250000** (which is the value I count up to).
I know that 18-bits is the right number, because **2^18 = 262144**, which is just large enough to store 250000. **r_Count** will never actually reach 262144, it will be reset before it gets that high.

Verilog Code – Debounce_Switch.v:

```
 1//////////////////////////////////////////////////////////////////////////////
 2// File downloaded from http://www.nandland.com
 3//////////////////////////////////////////////////////////////////////////////
 4// This module is used to debounce any switch or button coming into the FPGA.
 5// Does not allow the output of the switch to change unless the switch is
 6// steady for enough time (not toggling).
 7//////////////////////////////////////////////////////////////////////////////
 8module Debounce_Switch (input i_Clk, input i_Switch, output o_Switch);
 9
10  parameter c_DEBOUNCE_LIMIT = 250000;  // 10 ms at 25 MHz
11
12  reg [17:0] r_Count = 0;
13  reg r_State = 1'b0;
14
15  always @(posedge i_Clk)
16  begin
17    // Switch input is different than internal switch value, so an input is
18    // changing.  Increase the counter until it is stable for enough time.
19    if (i_Switch !== r_State && r_Count < c_DEBOUNCE_LIMIT)
20      r_Count <= r_Count + 1;
21
22    // End of counter reached, switch is stable, register it, reset counter
23    else if (r_Count == c_DEBOUNCE_LIMIT)
24    begin
25      r_State <= i_Switch;
26      r_Count <= 0;
```

```
27    end
28
29    // Switches are the same state, reset the counter
30    else
31      r_Count <= 0;
32  end
33
34  // Assign internal register to output (debounced!)
35  assign o_Switch = r_State;
36
37endmodule
```

Building the Project

Now build the project and let's take a look at the results.
First, take a quick look at the synthesis report to see how the tools synthesized our code.
We see much more usage than any previous projects, which makes sense, since now we have more functionality. Neat! Now program your board and make sure it's debouncing correctly.
Every push should now correspond to a toggle on the LED!

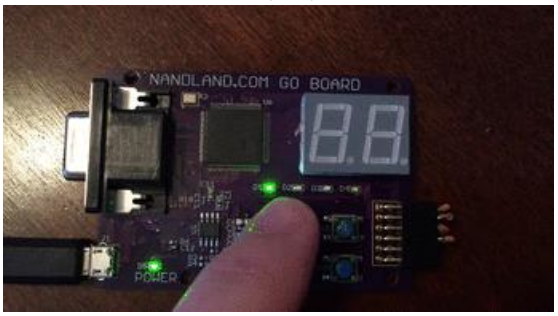
```
Resource Usage Report for Debounce_Project_Top

Mapping to part: ice40hx1kvq100
Cell usage:
GND             1 use
SB_CARRY        16 uses
SB_DFF          3 uses
SB_DFFSR        18 uses
SB_GB           1 use
VCC             1 use
SB_LUT4         32 uses

I/O ports:      3
I/O primitives: 3
SB_GB_IO        1 use
SB_IO           2 uses

I/O Register bits:                  0
Register bits not including I/Os:   21 (1%)
Total load per clock:
   Debounce_Project_Top|i_Clk: 1

Mapping Summary:
Total  LUTs: 32 (2%)
```

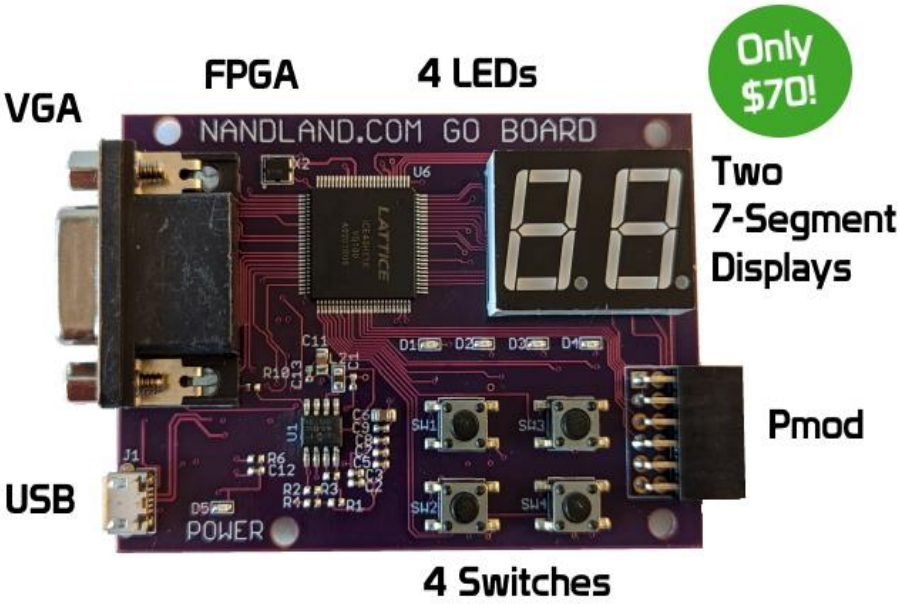


Pressing Switch 1 Toggles LED... Always!

Your Next Go Board Project: 7-Segment Displays

Go Board Tutorials

- Download and Install the FPGA Tools
- Project 1 – Your First Go Board Project
- Project 2 – The Look-Up Table (LUT)
- Project 3 – The Flip-Flop (AKA Register)
- Project 4 – Debounce A Switch
- Project 5 – Seven Segment Display
- Project 6 – How To Simulate Your FPGA Designs
- Project 7 – UART Part 1: Receive Data From Computer
- Project 8 – UART Part 2: Transmit Data To Computer
- Project 9 – VGA Introduction (Driving Test Patterns to VGA Monitor)
- Project 10 – PONG
- Go Board Support Stuff



Leave A Comment