Debouncing Circuit

About the file:

This file presents a comprehensive overview of my design, which involves implementing a denouncing circuit using the delay detection bouncing technique. It covers various aspects, including the circuit's architecture, RTL coding, synthesis, and formal verification. Additionally, the file delves into the concept of debouncing, providing an explanation of its principles and discussing different circuit implementation techniques. Furthermore, it offers a comparative analysis between the introduced architecture and an alternative approach, offering valuable insights into their respective strengths and limitations.

==========

Contents:

==========

Introduction:

Buttons or mechanical switches are commonly used as input devices in digital systems. When a button is pressed or released, it generates electrical signals that can be interpreted as digital logic levels (0s and 1s). However, due to the mechanical nature of buttons, they can introduce unwanted noise or bouncing effects.

Button bouncing refers to the rapid transition of the button's electrical contacts as they come into contact with each other or separate. This bouncing can cause multiple, rapid transitions of the signal from high to low or low to high within a short period, even if the button is pressed or released only once. This bouncing phenomenon can lead to erroneous or unreliable readings in digital circuits, as shown in Figure (1), where the input idle state is 0 and when the button is pressed, the input becomes 1.

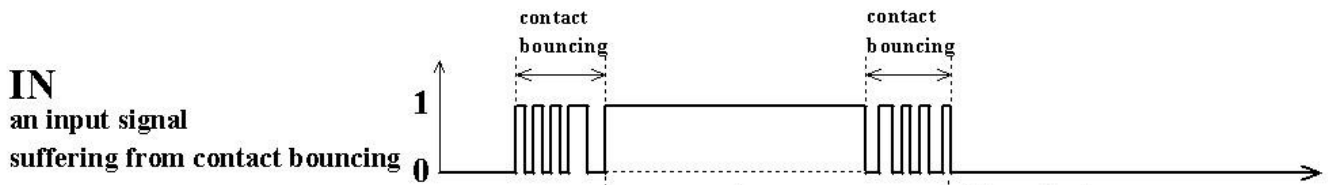

Figure (1): signal suffers from bouncing problem

==========

What is the denouncing circuit?

A denouncing circuit's goal is to reduce or completely remove the effects of button bouncing. It guarantees that for every button press or release event, only one clean and stable logic level is generated. By filtering out the bouncing noise, the denouncing circuit produces a consistent and reliable digital signal, as shown in Figure (2).

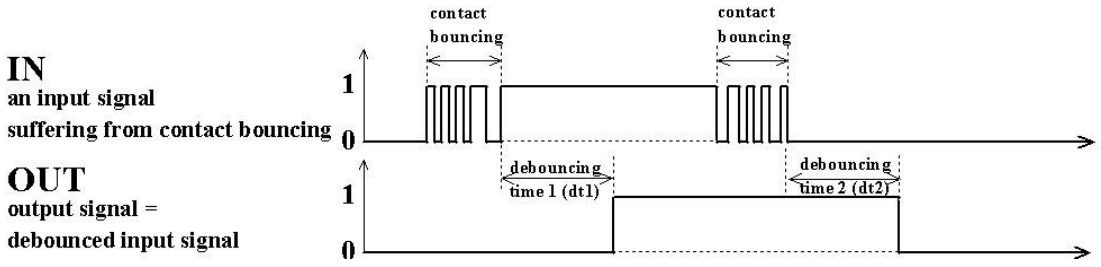

Figure (2): signal suffers from bouncing problem Vs signal after debouncing circuit

==========

denouncing circuit implementation techniques:

There are various techniques to implement a denouncing circuit; we will introduce the difference between two different techniques and then go into detail about one of them with its architecture, as it is the most basic and widely used technique. The two techniques are delay detection and early detection.

The delay detection debouncing technique involves introducing a delay or time interval before considering a transition as valid. The circuit waits for a predetermined amount of time, known as the debounce delay, when a switch is pressed or released before identifying the new stable state. Any bouncing or transient changes in the switch signal are ignored during the delay period. If the switch signal doesn't change after the delay, the transition is considered as valid, and the circuit responds accordingly. The delay detection method makes sure that transitions are stable by giving the mechanical switch enough time to stabilize and get rid of any bouncing effects.

Rather than waiting for a predetermined amount of time, **the early detection debouncing technique** focuses on identifying and responding to the switch signal's initial transition. Regardless of any further bouncing, the circuit responds immediately to the first detected transition when a switch is pressed or released. After capturing the initial transition, the circuit checks to see if the switch signal has stabilized in the new state by waiting a brief amount of time, known as the glitch filter time. The circuit operates as intended if the switch signal stays stable after the glitch filter time, indicating that the transition is valid. Otherwise, the bouncing is ignored, and it resets. The early detection method filters out bounces that happen soon after the initial transition while still responding rapidly to the user's input.

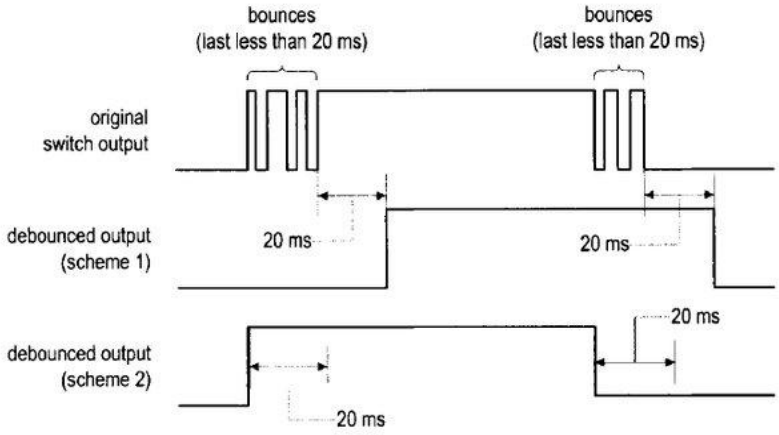

Figure (3): delay detection and early detection techniques

In Figure (3), we illustrate two different techniques for debouncing circuits: the delay detection technique (scheme 1) and the early detection technique (scheme 2). Let's examine their characteristics:

In the delay detection technique, the output signal considers a high transition valid only if the original signal remains stable for a specific duration known as the debounce delay. In this case, we assume a debounce delay of 20 ms. Similarly, the low transition at the end of the waveform follows the same principle.

In contrast, the early detection technique identifies a high transition as valid during the initial transition introduced by the input. It then waits for a predetermined time, referred to as the glitch filter time (assumed to be 20 ms), to check if the captured signal value remains stable. In figure (3), the initial high transition is captured, and after the glitch filter time elapses, another high transition is captured, confirming its validity. The same process occurs for the low transition at the end of the waveform.

In conclusion, the early detection technique captures the initial transition quickly and checks to see if the switch signal stabilizes within a short period of time, effectively filtering out bounces that happen soon after the initial transition, while the delay detection technique introduces a time delay before considering a transition as valid, allowing the switch signal to settle over time. Both methods use different strategies to deal with switch bouncing, but they both aim to produce reliable and stable signal transitions in debouncing circuits. The specific application requirements and desired debouncing circuit behavior determine which technique should be used.

==========

Introduced architecture for the delay detection technique:

Note that, this is an introduced architecture that I used to implement the denouncing so you can search for more different implementations

As we said before this technique involves introducing a delay or holding period after the button state changes which is known as delayed detection technique. During this delay period, any bouncing effects are ignored, and the circuit waits for the button signal to stabilize. Once the delay period elapses without any further bouncing, the denouncing circuit registers the button's final state. The inputs and outputs of the block are easily identifiable, as depicted in Figure (4) and summarized in Table (2). Also, Table (1) summarizes the parameters used in my design.

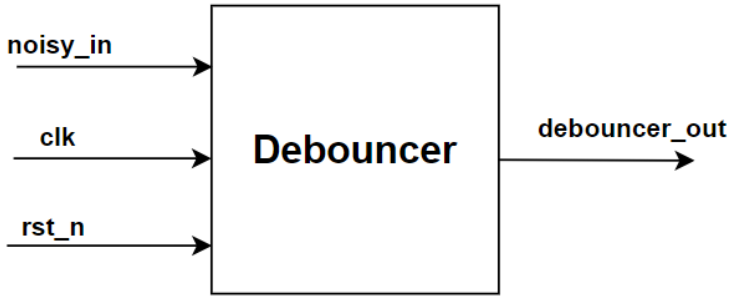

Figure (4): denouncing circuit block diagram

parameter	description
num_stages	the number of flip/flops used in the synchronizer
counter_final_value	the required final value for the counter in the timer circuit that corresponding to the needed delay

Table (1): parameters of denouncing circuit

signal	width	direction	description
clk	1	input	Positive edge system clock
rst_n	1	input	Asynchronous negative edge system reset
noisy_in	1	input	Input signal suffering from button bouncing
debouncer_out	1	output	Debounced output signal

Table (2): inputs and outputs of denouncing circuit

When considering the circuit architecture, it becomes apparent that several components are necessary for its proper functioning. Firstly, a finite state machine (FSM) is required to facilitate the transition from one state to another based on the input signal affected by the bouncing issue. Additionally, a timer circuit is essential to provide the FSM with the necessary delay information, enabling it to make informed decisions regarding transitions, as the system relies on the delay detection period.

Addressing the problem of buttons and switches that we aim to resolve with this denouncing circuit, it is crucial to incorporate a synchronizer at the input of the original signal. This synchronization step is vital to mitigate the bouncing effect. Without it, the direct utilization of the button signal can lead to metastability, especially when the button signal undergoes changes near the clock edge.

Hence, as depicted in Figure (5), the denouncing circuit consists of three key blocks: the FSM, synchronizer, and timer circuit. Each plays a significant role in achieving reliable and accurate debouncing circuit functionality.

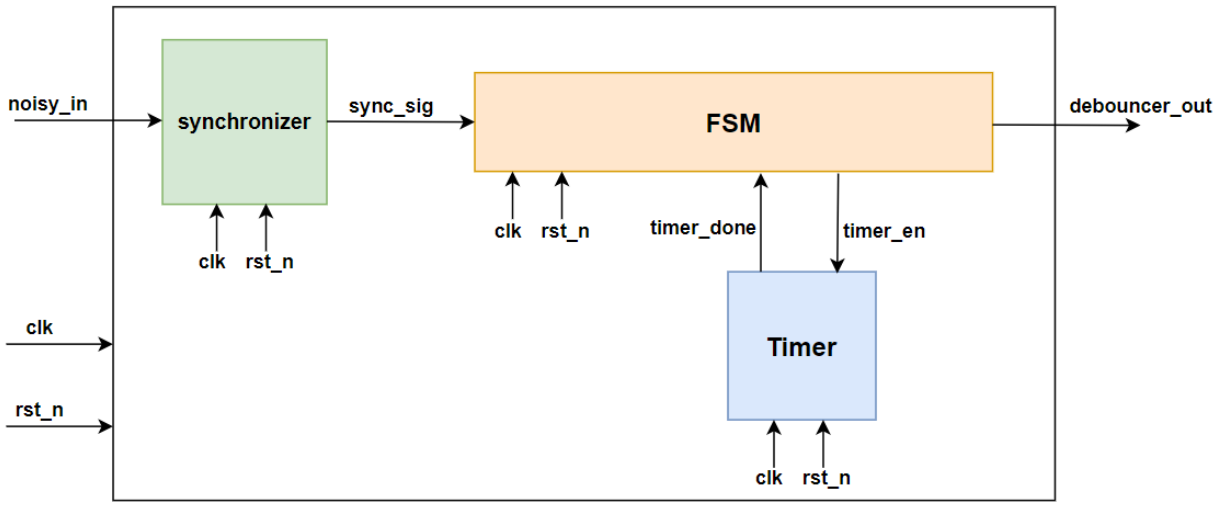

Figure (5): debouncing circuit architecture

Synchronizer:

As mentioned before, Its primary purpose is to avoid metastability issues that can occur when two asynchronous signals, such as the button signal and the clock signal, interact. Metastability is a phenomenon that can occur in digital circuits when a signal changes near the edge of a clock cycle. It can result in an uncertain or unpredictable output, which may lead to incorrect behavior of the circuit. When a button signal is used directly without synchronization, metastability can occur if the button signal changes near the clock edge.

Its implementation is simply a number of flip-flop consecutive stages, such as the 2-stage synchronizer, which has only two flip-flops. The inputs and outputs of the block are easily identifiable, as depicted in Figure (6) and summarized in Table (4). Also, Table (3) summarizes the parameters used in my design.

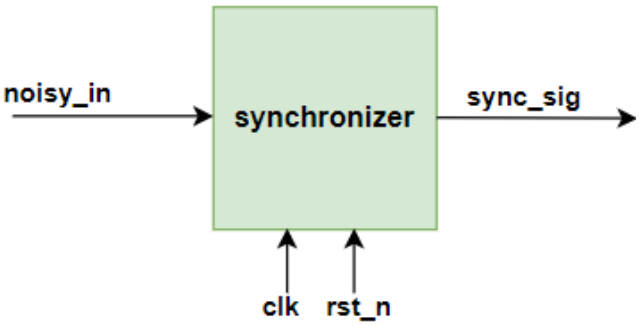

Figure (6): synchronizer block diagram

parameter	description
num_stages	the number of flip/flops used

Table (3): parameters of the synchronizer

signal	width	direction	description
clk	1	input	Positive edge system clock
rst_n	1	input	Asynchronous negative edge system reset
in_sig	1	input	the input signal suffering from button bouncing without synchronization
out_sig	1	output	the input signal suffering from button bouncing after synchronization

Table (4): inputs and outputs of the synchronizer

Timer:

A timer circuit is employed to measure the desired delay by utilizing a counter to convey the necessary delay information to the Finite State Machine (FSM). For example, with a clock frequency of 100 MHz and a corresponding clock period of 10 ns, the target delay is assumed to be 1000 ns. Therefore, the counter is configured to count 100 clock cycles, ranging from 0 to 99, in order to achieve the specified delay duration.

When the FSM issues the timer_en enable signal, the timer circuit begins counting for the delay. When the timer reaches the desired delay, it sends a flag to the FSM informing it that the delay detection period has passed. The inputs and outputs of the block are easily identifiable, as depicted in Figure (7) and summarized in Table (6). Also, Table (5) summarizes the parameters used in my design.

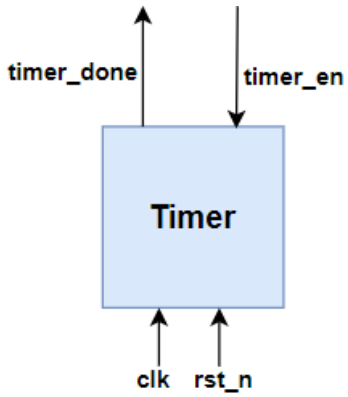

Figure (7): timer block diagram

parameter	description
counter_final_value	the required final value for the counter corresponding to the needed delay

Table (5): parameters of the timer

signal	width	direction	description
clk	1	input	Positive edge system clock
rst_n	1	input	Asynchronous negative edge system reset
timer_en	1	input	enable signal to start counting
timer_done	1	output	output flag tells the FSM whether it finish counting the targeted delay or not

Table (6): inputs and outputs of the timer

FSM:

A Finite State Machine (FSM) is a mathematical model used to represent and control system behavior. It consists of a finite number of states, transitions between these states, and actions associated with the transitions. The role of an FSM is to define the behavior and logic of a system by responding to inputs and changing states accordingly. It has two types, mealy FSM where the output depends on both input and current state or moore where the outputs depends only on the current state. We use moore FSM in our system.

FSM takes the synchronized original signal that outputs from the synchronizer and timer_done flag from the timer circuit to know whether the delay detection period has passed or not after it requests to start the delay calculations through its output signal timer_en that goes to the timer circuit. FSM also outputs the debounced output signal after filtering the bouncing effect. The inputs and outputs of the block are easily identifiable, as depicted in Figure (8) and summarized in Table (7).

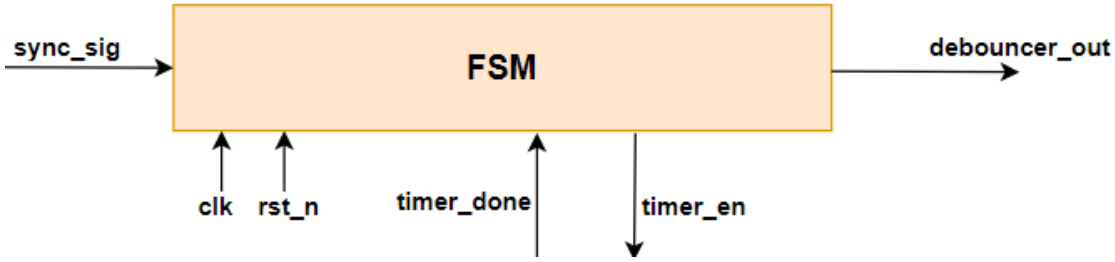

Figure (8): FSM block diagram

signal	width	direction	description
clk	1	input	Positive edge system clock
rst_n	1	input	Asynchronous negative edge system reset
sync_sig	1	input	Input signal suffering from button bouncing after synchronization
timer_done	1	input	input flag tells the FSM whether the timer finish counting the targeted delay or not
debouncer_out	1	output	Debounced output signal
timer_en	1	input	output enable signal to the timer to start counting

Table (7): inputs and outputs of the FSM

To identify the FSM operation, we need to look at its state diagram, which is depicted in Figure (9):

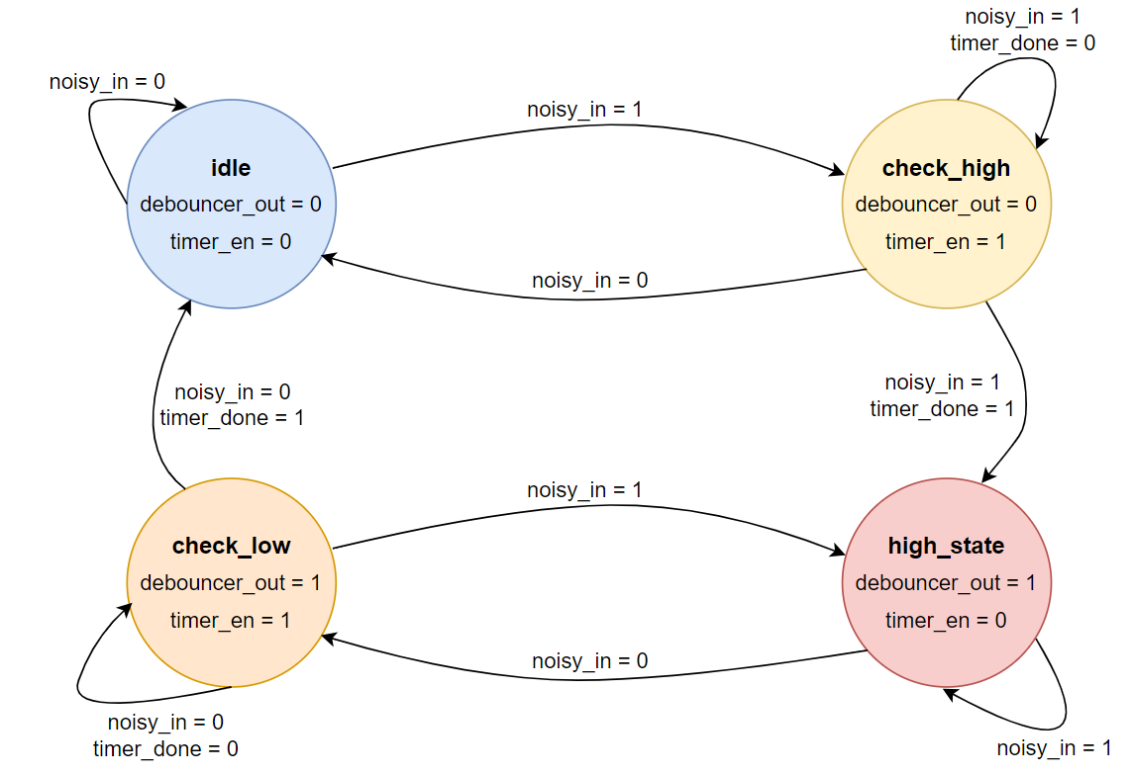

Figure (9): FSM state diagram

idle

- This is the idle state where all outputs are zeros, and if the noisy input signal is still zero, we will still be in the idle state; otherwise, we will move to the check_high state.

check_high

- This is the state where we check if the high input signal will stay high for the targeted delay or not, so the denouncing_out is low as we still check if it is a high signal or just a glitch, but timer_en becomes high to enable the timer circuit to start calculating the delay.
- If the noisy input signal becomes low, it is just a glitch, so we move again to the idle state.
- If the noisy input signal is still high but the timer_done input from the timer circuit is low, the targeted delay has not passed yet, so stay in the check_high state as the check has not completed yet.
- If the noisy input signal is still high but the timer_done input from the timer circuit is high, the signal will stay high during the targeted delay, so we will move to the high state after ensuring that the input high signal is a real signal and not a glitch.

high_state

- This is the state where we output high denouncing_out but timer_en becomes low as we don't need to calculate any delays here.
- If the noisy input signal becomes low, we move to the check_low state.
- If the noisy input signal is still high, we stay in the check_high state.

check_low

- This is the state where we check if the low input signal will stay low for the targeted delay or not, so the denouncing_out is high as we still check if it is a low signal or just a glitch, but timer_en becomes high to enable the timer circuit to start calculating the delay.
- If the noisy input signal becomes high, it is just a glitch, so we move again to the high state.
- If the noisy input signal is still low but the timer_done input from the timer circuit is low, the targeted delay has not passed yet, so stay in the check_low state as the check has not completed yet.
- If the noisy input signal is still low but the timer_done input from the timer circuit is high, the signal will stay low during the targeted delay, so we will move to the idle state after ensuring that the input low signal is a real signal and not a glitch.

==========

Simulation results

The simulation was performed using Modelsim, utilizing the provided Verilog testbench code available on the GitHub link. The clock frequency used during the simulation was set at 100 MHz, corresponding to a 10 ns period. The desired delay was 1000 ns, equivalent to 100 clock cycles. Consequently, the counter was configured to count from 0 to 99 to achieve this delay. Additionally, a 2-stage synchronizer was employed to ensure proper synchronization of the signals.

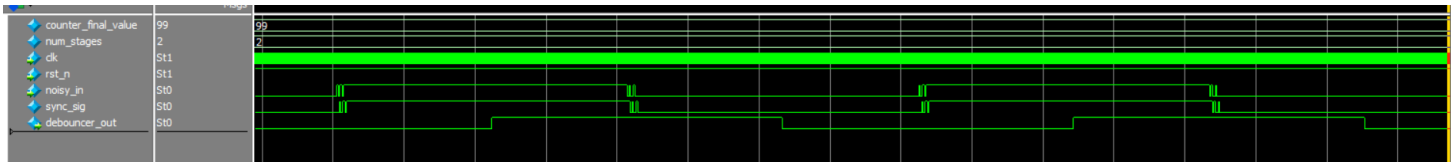

Figure (10): denouncing circuit simulation results

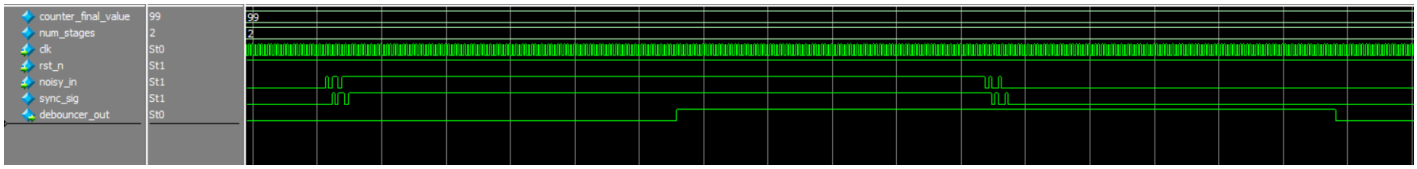

Figure (11): denouncing circuit simulation results (zoomed)

==========

Synthesis results

The synthesis process was performed using the Design Compiler tool. System clock frequency is 100 MHz. Our target is to minimize the area. Synthesized netlist schematic is shown in Figure (15).

Spec	name
technology	tsmc_130nm
wire model	tsmc13_wl30

Table (8): technology specification

Constraints	value/name
create_clock	period: 10 ns
set_clock_uncertainty -setup	0.2 ns
set_clock_uncertainty -hold	0.1 ns
set_clock_transition -rise	0.05 ns
set_clock_transition -fall	0.05 ns
set_clock_latency	0
set_input_delay	0.5*clock period
set_output_delay	0.5*clock period
set_driving_cell	BUFX2M
set_load	0.5 pF
set_max_area	0
set_dont_touch_network	clk

Table (9): used constraints

outputs	value
Total cell area	581.289811 um^2
Total area	12901.352158 um^2
Combinational area	221.219603 um^2
Buf/Inv area	23.534001 um^2
Non-combinational area	360.070208 um^2
Net Interconnect area	12320.062347 um^2
Noncombinational area	360.070208 um^2
Net Interconnect area	12320.062347 um^2
Switching power	9.03e-03 mW
Internal power	3.11e-02 mW
Leakage power	3.88e+05 pW
Total power	4.05e-02 mW
Worest setup slack	3.31 ns
Worest hold slack	0.52 ns

Table (10): outputs values from synthesis

==========

Formal verification results

Formal verification is done successfully using the Formality tool as shown in figure(12) by the TCL script available at github link.

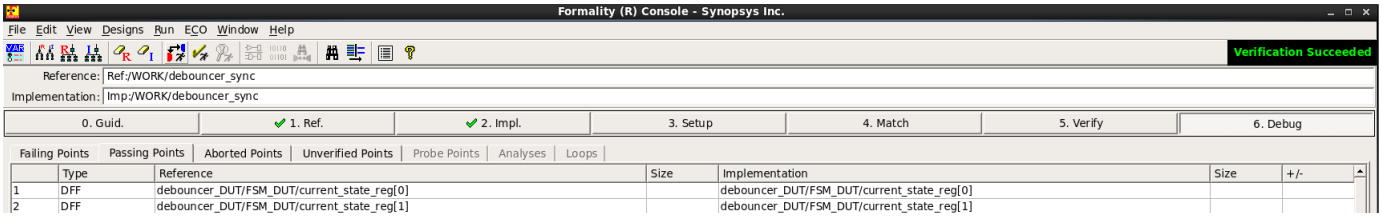

Figure (12): formal verification results

==========

Compare with another delay detection implementation:

When considering the fundamental idea behind the delay detection technique, the detection period is determined by counting the number of clock periods required for the desired delay. One might initially assume that this delay can be achieved by introducing a series of registers equal to the number of clock cycles needed for the delay. In this approach, the FSM and timer circuit can be represented by a consecutive set of registers, and the value of each register is checked. When all registers hold the same value (either 1 or 0), it ensures that the signal remains stable for the detection delay period represented by the number of registers.

While this approach holds true, it may not be the most efficient choice for longer delay detection periods. The number of registers required would exceed that needed for the counter in the timer circuit. For instance, let's consider a clock frequency of 100 MHz with a corresponding clock period of 10 ns. Assuming a target delay of 1000 ns, which corresponds to 100 clock cycles, using the register delay concept would necessitate 100 registers for delaying the signal and checking their values. However, with the timer circuit, a counter can be configured to count 100 clock cycles from 0 to 99, requiring only 10 registers for the counter.

You may raise another question regarding the need for the FSM when using the timer circuit. It is true that employing the timer circuit still requires the FSM discussed earlier, so the overall area difference may not be substantial. To obtain a conclusive answer regarding the total area, it is necessary to implement both architectures and utilize appropriate tools to evaluate the resulting area, as demonstrated below.

Area visual comparison:

1. **Using Vivado:**

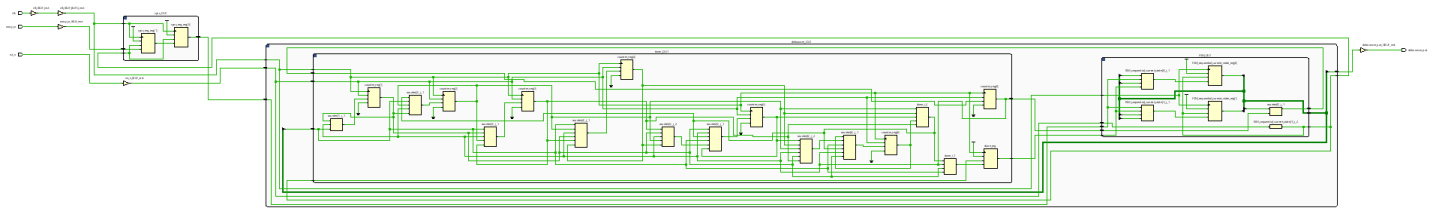

Figure (13): Synthesized netlist (FSM and timer implementation) using Vivado

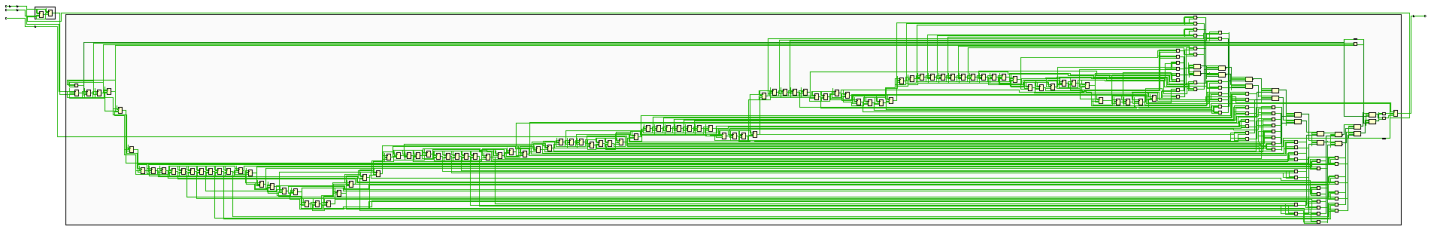

Figure (14): Synthesized netlist (delayed registers implementation) using Vivado

By using Vivado to check the number of needed cells for both implementations, we found that:

Using FSM and timer	Using delay registers
35 cells	198 cells

Table (11): comparison between the two implementation using Vivado

2. Using design compiler:

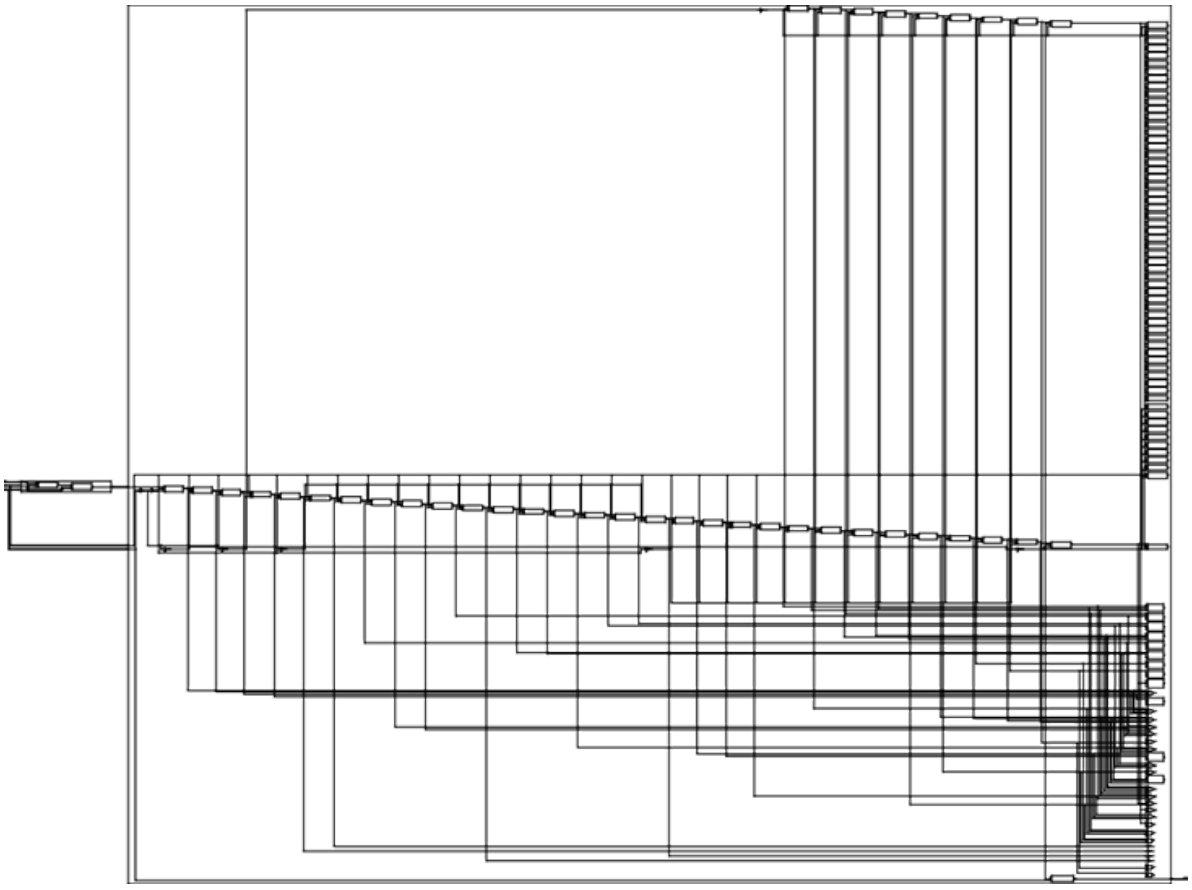

Figure (15): Synthesized netlist (FSM and timer implementation) using Design Compiler

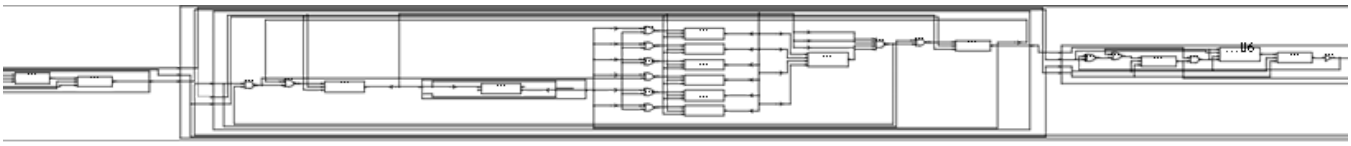

Figure (16): Synthesized netlist (delayed registers implementation) using Design Compiler

By using synopsys design compiler to check the total are for both implementations after synthesis, we found that:

	Using FSM and timer	Using delay registers
total cell area	581.289811 um^2	3486.562080 um^2
total area	12901.352158 um^2	88886.998970 um^2

Table (12): comparison between the two implementation using design compiler

==========

Github project link:

https://github.com/FatmaAli99/Debouncer