Tutor: Dr N. Boulgouris

Author: Htet Paing Aung

Student ID: 2361266

Email: 2361266@brunel.ac.uk

Title: Design and Implementation of Sequence Detector on FPGA with Nexys Constraints

Module: EE3635 Embedded Systems

Date: 4. Dec. 2024

Pages: 15 main page and 2 outline

Keywords: Vivado, VHDL, Embedded systems, Digital security, XILINX, FPGA programming, Hardware description languages, shift register , flag parallel processing

Department of Electronic and Computer Engineering

Brunel University London

Table of Contents

Figures

Introduction

This project focuses on developing a robust FPGA-based lock system using advanced digital design techniques, including clock dividers, finite state machines (FSMs), debounce mechanisms, and shift registers. The system aims to deliver precision, stability, and inclusivity, addressing challenges such as mechanical switch bouncing and timing inconsistencies. By employing a three-stage shift register debounce mechanism and synchronized clocking, the system ensures noise-free operation and reliable button press detection.

At its core, the system features a hybrid FSM that manages button press counting, sequence detection, and lock activation. The clock divider reduces a high-frequency clock signal into a slower, synchronized clock, enabling seamless operation for tasks like debouncing, LED updates, and flash signalling. These features ensure accurate and responsive performance, making the system suitable for a variety of applications, including emergency scenarios and general-purpose use.

Inclusivity was a key consideration, ensuring usability for diverse users, such as those with slower response speeds or visual impairments. Features like responsive debouncing and clear visual indicators provide accessibility and adaptability. The system also prioritizes resource efficiency, with parallel processing and optimized power consumption, ensuring fast, reliable, and energy-efficient operation. This report outlines the system's theoretical foundation, implementation approach, challenges, and results, demonstrating how FPGA-based designs can create reliable, user-friendly solutions for modern digital applications.

Theory background

Clock divider

As the counter is based on flip flick to generate the clock by assigning in code level, the frequency achieved will always the divided amount of the original. Along with that, the frequency divider from a paper in reference is used by integrating half amount of original 100MHz as in figure 1 below. Moreover, the output frequency is further used to calculate the operating cycles and operating time in sec in below sessions.

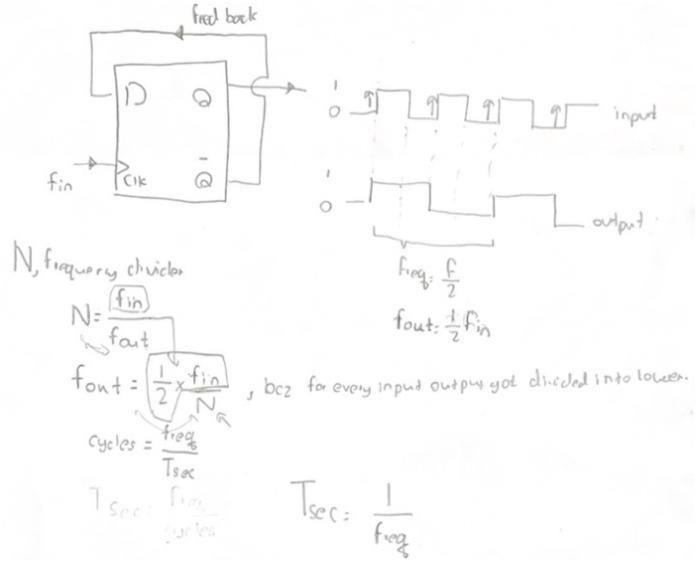


Figure 1 clock divider

Finite Sate Machine

Based on the two failure implementation system pipelines, the best robust hybrid finite state machine is constructed with the conventional FSM adder counter that count from 0 to 155 with assigning btnc as an ideal reset. Next, the flag is used in code level to implement the lock on 11th input attempt. After that, the sequence detection with binary symbol encoding is used to detect a specific pattern (00,01,01,10,11) and display an unlock flash pattern as Shown below figure 2.

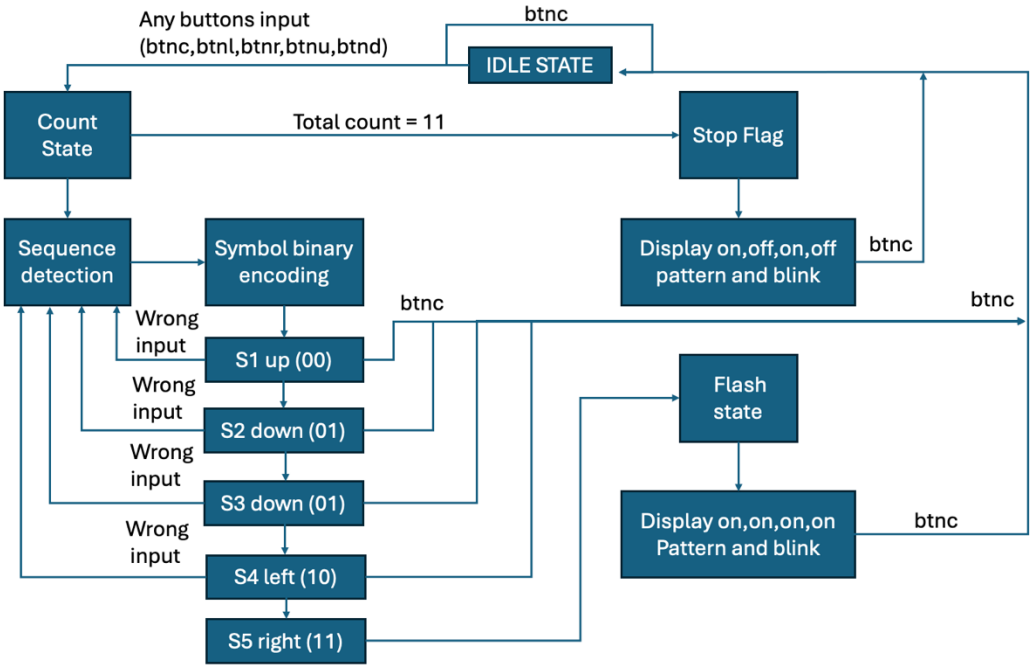
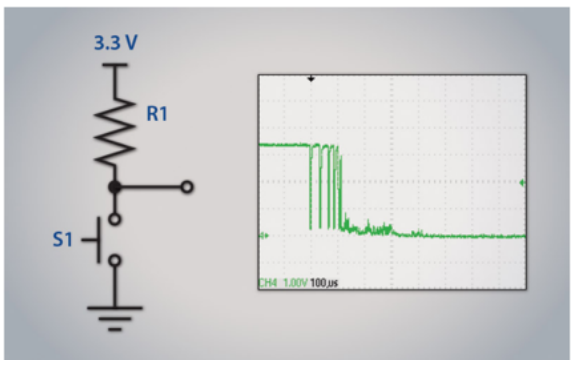


Figure 2 State machine

Debouncing

As the two most common types of debounce strategy are shift register and threshold delay debounce and the debounce arise from the mechanical switch nature as shown in below figure 3. Moreover, the below table show the compares between normal threshold delay debounce and three stage shifts register debounce bas e on mechanism, state recognition, bouncing detection, implementation simplicity, memory usage, processing overhead and use cases.



Aspect	Threshold Delay Debounce	Three-Stage Shift Register Debounce
Core Concept	Monitors the button state over time using a counter to detect stability.	Uses a shift register to track button state across multiple sampling intervals.
Mechanism	Increments a counter for consistent readings and resets it when readings differ.	Shifts the sampled state into a register, determining stability based on register bits.
State Recognition	Stable state is confirmed when the counter reaches a predefined threshold.	Stable state is confirmed when all bits in the shift register are consistent (all 1s or 0s).
Bouncing Detection	Transient state changes reset the counter, preventing unstable states from being registered.	Mixed bits in the shift register indicate a bouncing state.
Implementation Simplicity	Easier to implement with basic logic and a counter.	Slightly more complex due to the need for a shift register.
Memory Usage	Minimal memory (a counter variable).	Requires a shift register, which may use more memory depending on its size.
Processing Overhead	Lower processing overhead due to simple logic.	Slightly higher overhead due to bit-shifting operations.
Use Case	Suitable for systems with constrained resources or simple button operations.	Ideal for systems requiring robust debouncing for high-frequency signals or noise.

Figure 3 Debounce two types

Shift Register

As shift register is a powerful technique that drive the modern digital parallel processing for telecommunications and AI processing applications for FPGA programming with the use of the sequential flip-flops design, it is utilized in dealing with switch bouncing. Interestingly, the data input clock pulse and data output are the basic operation of the shift register as shown in below figure. Interestingly, the three common types of shift registers are Serial-in serial-out (SISO), Serial-in parallel-out (SIPO) and Parallel-in serial-out (PISO) as shown on below figure. Noticeably, the lock system uses serial in serial out (SISO) shift register as shown in below figure 4.

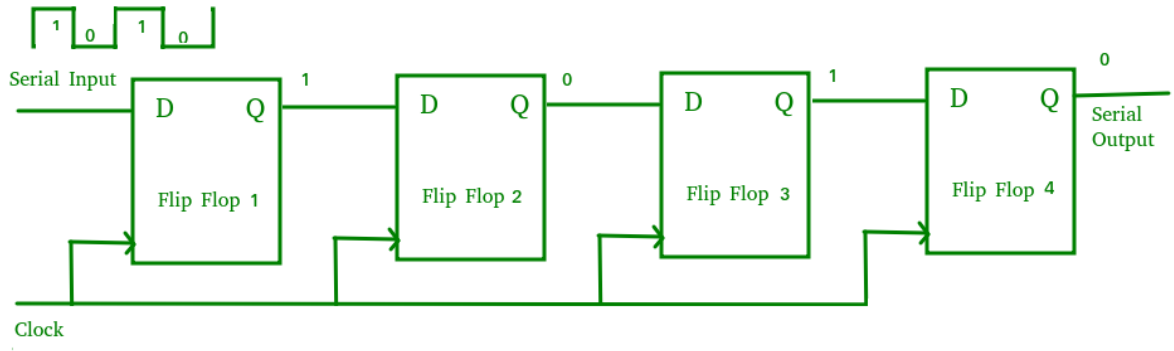


Figure 4 SISO shift register

Flags

Flag is implemented as a signal or variable that indicates the occurrence of a on and off lock pattern. Flags are essential for controlling the flow of processes and managing state transitions in hardware descriptions and it is used thought the implementation.

```
signal counter : integer := 0;
signal threshold_reached : std_logic := '0';

process(clk)
begin
    if rising_edge(clk) then
        if counter = 10 then
            threshold_reached <= '1'; -- Set flag when counter reaches 10
        else
            threshold_reached <= '0';
        end if;
        counter <= counter + 1;
    end if;
end process;
```

Figure 5 FLAG

Equity Diversity and inclusion

In developing this FPGA-based system, equality, diversity, and inclusion were fundamental considerations throughout the design process. By incorporating features that address the needs of diverse user groups, such as elderly individuals or those requiring fast emergency responses, we ensured that the system is accessible and adaptable for any situation. Implementing features like a responsive debounce mechanism, a three-stage shift register, and visual indicators tailored to assist users with varying paces and vision capabilities reflects the commitment to inclusive design.

Moreover, the use of synchronized clocks and optimized logic to provide a seamless and reliable user experience demonstrates the well intent to accommodate diverse skill levels, from beginners to advanced users. The focus on creating intuitive interaction through a clean and responsive buttons design ensures that the system can be easily understood and operated by individuals from different educational and technical backgrounds.

By designing with universally beneficial features such as stable counters, clear visual indicators, and reliable button noise filtering, a system that embodies fairness in usability, fostering confidence and trust among all users emerged. Additionally, the adaptability of the system ensures that it can be applied in various environments, ranging from educational settings to high-stress response emergency scenarios, further highlighting dedication to inclusivity. The systems that prioritize user needs, break barriers and promote equal access to technology as all technology should empower all individuals, and through thoughtful engineering and inclusive practices.

Although, the system can cover all diversity and inclusion it is not well suited for peoples with disability as the current system does not support an extra feature for disabled people as implementation upgrade need more mechanical and design involvement. As for future practice and upgrade, the mechanical switch design tailor to disabled peoples with implementation IOT by using HTTPS or reliable protocols establishment to transmit data to cloud or in house server for sensing and unlocking lock system can be further implemented for equity diversity and inclusion.

Architecture of system development pipeline

The overall system uses bottom-down approach and parallel system implementation with according to the abstracted description as show in diagram as below figure 6. Initially, the five buttons with their corresponding led displays are constructed which represents 4123 as (butnu, butnd, butnl, butnr). Secondly, the counter with the total number of counting display and central button as a rest is implemented.

By taking advantage of establishing code, the display of on and off pattern on every 11^{th} total number of four button combined input is implemented. Next, the stop flag is implemented to freeze and display the on off lock LED pattern when the total count of all buttons inputs is 11. Along with that, the parallel implementation with the sequence logic as in below figure 6 is implemented for displaying all led when the correct number of sequences is entered with the same clock and counter setup as the existing count and lock system.

Followingly, the two well established correct pattern detection and the counter with lock system are combined to create a lock system that display all on when the input sequence is correct (41123, up down down left right represent btnu btnd btnd btnl btnr) and lock and display the on and off pattern when the total count become 11. Next, the flash state is implemented in the counter to flash the two led display patterns by establishing new flash signal. As for better button response for emergency situations and elders friendly in mind, the shift register with delay bounce is implemented in the final step of FPGA system development pipeline to prevent counting on a single button press. Furthermore, the step-by-step executable code block can be seen in figure 6 below and the detail code progressions can be seen in the Appendix 1.

- the sequence btnu, btnd, btnd, btnl, btnr , any other 5 buttions
- the sequence any other 5 buttions ,btnu, btnd, btnd, btnl, btnr ,
- the sequence any 1 button input, btnu, btnd, btnd, btnl, btnr , any other 4 button inputs.
-the sequence any 2 button input, btnu, btnd, btnd, btnl, btnr , any 3 button inputs,
- the sequence any 3 button input ,btnu, btnd, btnd, btnl, btnr , any 2 button input,
-the sequence any 4 buttons input, btnu, btnd, btnd, btnl, btnr , any 1 button input,

Using Vivado, write VHDL code for the implementation of a sequence detector in VHDLusing push buttons. Four push buttons should be used for entering four input buttons(btnu,btnd, btnl, btnr).Further, the push button btnc in the middleshould be used for initialisation. •Up to 10 inputs could be entered after pressing the initialisation push button. When the sequence (btnu,btnd, btnd, btnl, btnr) is entered, then the LEDs should start flashing. Please note that the right sequence of symbols need not necessarily be entered immediately after the initialisation push button is pressed. For example, the sequence "btnu,btnd, btnl, btnr,btnr,btnu,btnd, btnd, btnl, btnr" should be able to activate the flashing of the LEDs.•If inputs have been entered but the right sequence of symbols has not appeared yet, then the system should lock and the LEDs should show the following predefined pattern:on, off, on, off, on, off, on, off,on, off, on, off.•If the system is locked, the user will need to press the initialisation button in order to be allowed to start entering new symbols

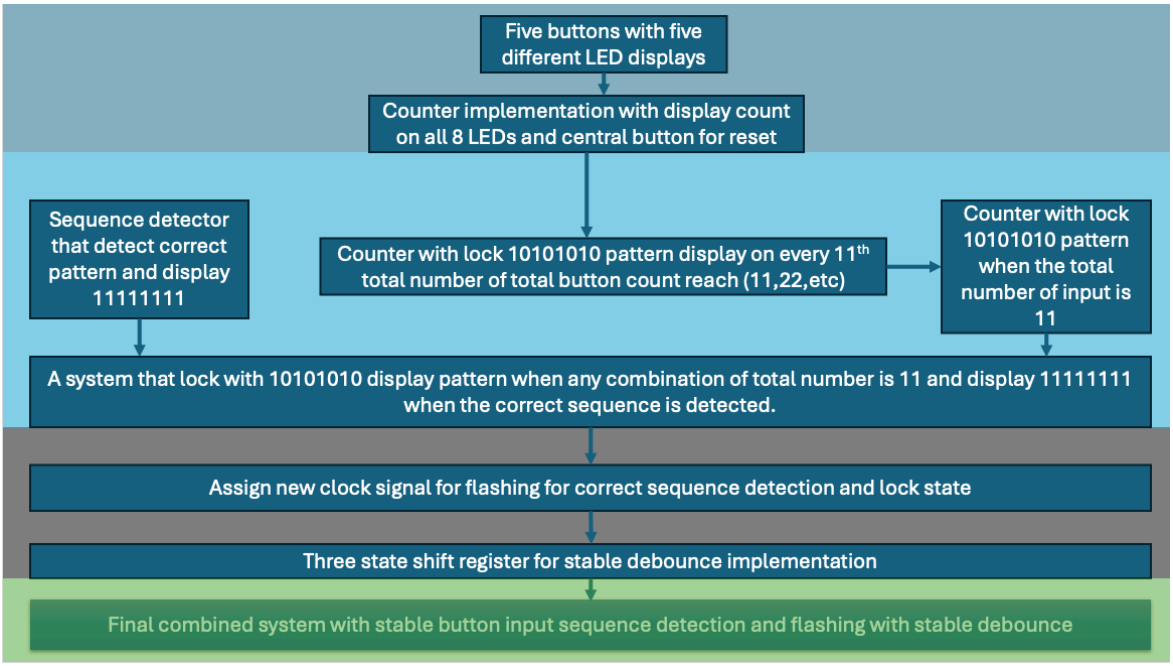


Figure 6 System pipeline architecture

Implementation Approach with detail component and logic

Implicit State Machine Detail break down

Clock Divider

Clock divider frequency

In the final design, the 100MHz original clock cycle is converted into 16.67Hz with 0.06s and 277.83cycles per sec operating slow clock for Debouncing, Counter sequence detection, updating LED patterns, button press counting and synchronizing system logic by using the constant of 3,000,000. Noticeably, the slow clock frequency is programmed with well-tuned fast operating cycles with edge detection and three stage delay logic to ensure stable and noise free transitions for fast responsive buttons for emergency fast input.
Moreover, the debounce logic method with shift register is reliable as it remains with-ought to count when the button is pressed which enable suitable input speed for all paces.

Interestingly, the flash clock is achieved with 10,000,000 constants which provide 5Hz with 0.2 operating speed and 25 operating cycles per sec as shown in below figure 7 by using the formulas as shown below. As for functionality wise, the flash clock is used in controlling blinking pattern rate of LEDs when the correct sequence is detected, creating an alternative pattern when counting steps and enhance visual with the right pattern rate to display flash, on-off datively for elderly with weak eyesight.

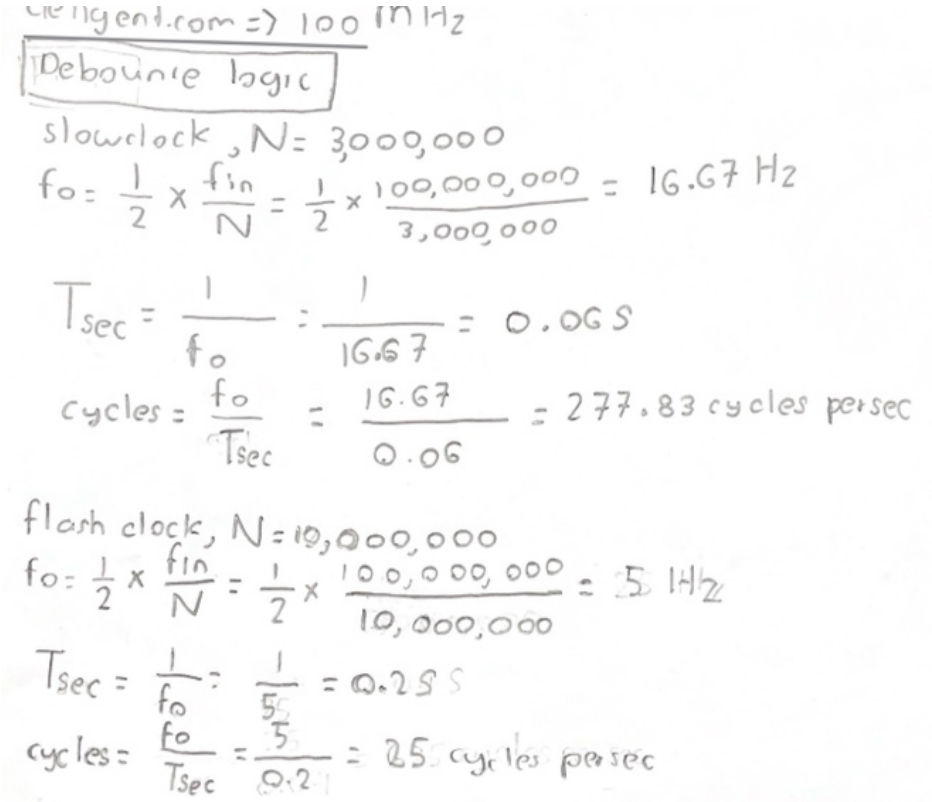


Figure 7 Frequency calculations

Clock divider variable components and logits

The rising edge (clk) ensure the flow execute on every upward edge to ensure synchronization and avoid potential glitches. Moreover, the clock count and flash count keep trach of how many times cycles occurred by assigning integer values as zero in architecture behaviour section. Furthermore, the clock divider -1 and the flash divider -1 further ensure the toggle happen on the correct stage 0 to N to N-1 (N=clock divider integer cycle).

To implement activate the flip flop condition of the clock the slow clock and the inverse version of it is equalled together to create 1 and 0 or 0 and 1 to activate the slow clock for later use and set the initial clock count as 0. In the final step of both clock count and flash count logic, the else is declared with clock count equal with clock count plus one enable the clock and flash counting to happen as shown in below code figure and the detail code can be found in final code implementation in Appendix 1.

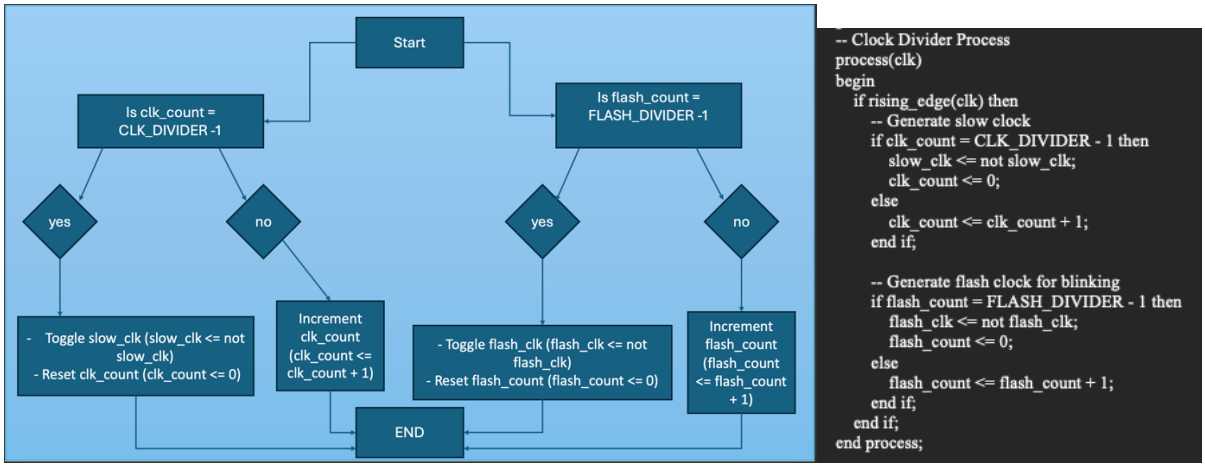


Figure 8 Clock divider logic

Debounce

Overall, the denouncer is used as a three-stage delay chain that enable a stable delay to pass through with edge detection logic with the use of slow clock for syncing with time scale for appropriate button input as show in in below figure 9. In the entity level, the Denouncer is declared as clk for slow clock and input for buttons inputs with output as debounced and cleaned signals. As for architecture, the signal is declared to delay1, delay2 and delay3 with std logic declaration for initiation. In the Process for cleaning noise, the three delay stages is used in below figure 9.

Initially, the delay 1 captures the noisy input that arise from mechanical switch in the current rising edge and delay 2 holds and filter the value of unwanted delay 1 noisy signal from previous clock cycle. At last, the delay 3 holds the value of delay 2 from two clock cycles. As a result, the unwanted noisy signal as in below figure switch bouncing got eliminated. Noticeably, the output stage is declaring for the trigger output to trigger only when the input has stabilized for at least a period.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debouncer is
    Port (
        clk    : in std_logic;
        input  : in std_logic;
        output : out std_logic
    );
end Debouncer;

architecture Behavioral of Debouncer is
    signal delay1, delay2, delay3 : std_logic := '0';
begin
    process(clk)
    begin
        if rising_edge(clk) then
            delay1 <= input;
            delay2 <= delay1;
            delay3 <= delay2;
        end if;
    end process;

    output <= delay2 and not delay3; -- Detect stable rising edge
end Behavioral;
```

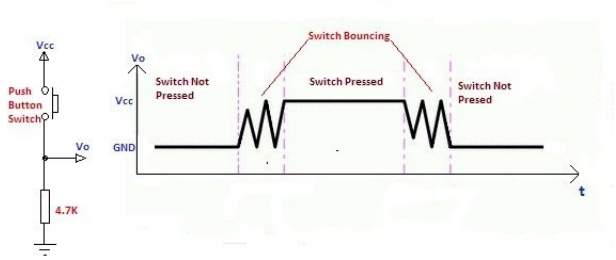


Figure 9 Shift register

Interestingly, the below bouncing delay figure further illustrates the debounce output occur only on the fifth clock cycle when the delay 1 and delay 2 have one as the code logic states. Moreover, the bouncing filtering happen in 2 to 4 clock cycles as show in below figure 10.

Clock Cycle	Raw Input (input)	delay1	delay2	delay3	Debounced Output
1	0	0	0	0	0
2	1 (bounce starts)	1	0	0	0
3	0 (bounce)	0	1	0	0
4	1 (bounce ends)	1	0	1	0
5	1 (stable)	1	1	0	1
6	1	1	1	1	0

Figure 10 3 stage debounce table

In the main state machine, the denouncer is instantiated in the CombinedSystem architecture as show in in below figure. Generally, the debouncer take the raw button inputs and produce debounced outputs. Also, the debounced five buttons are used in throughout the state machines in resetting, counting, recording button sequences and detecting the target sequence as shown in Appendix 1 final codes and below code figure 11.

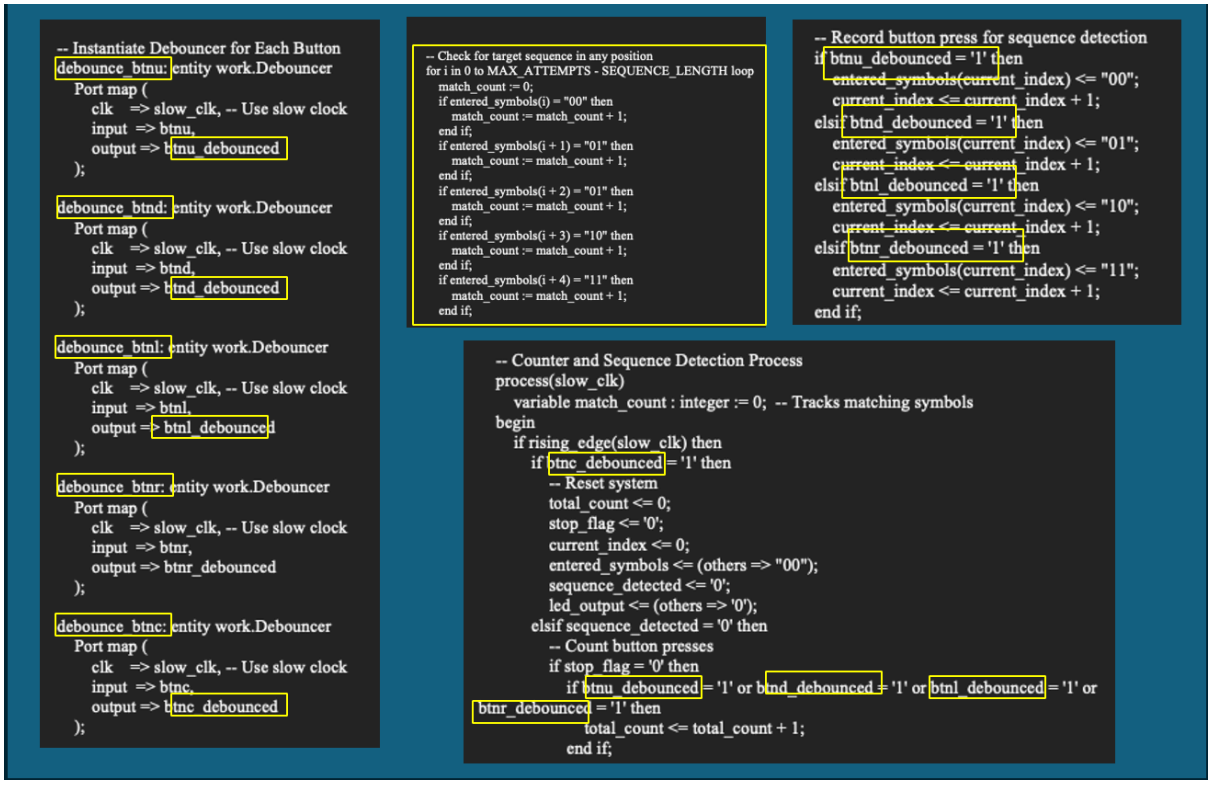


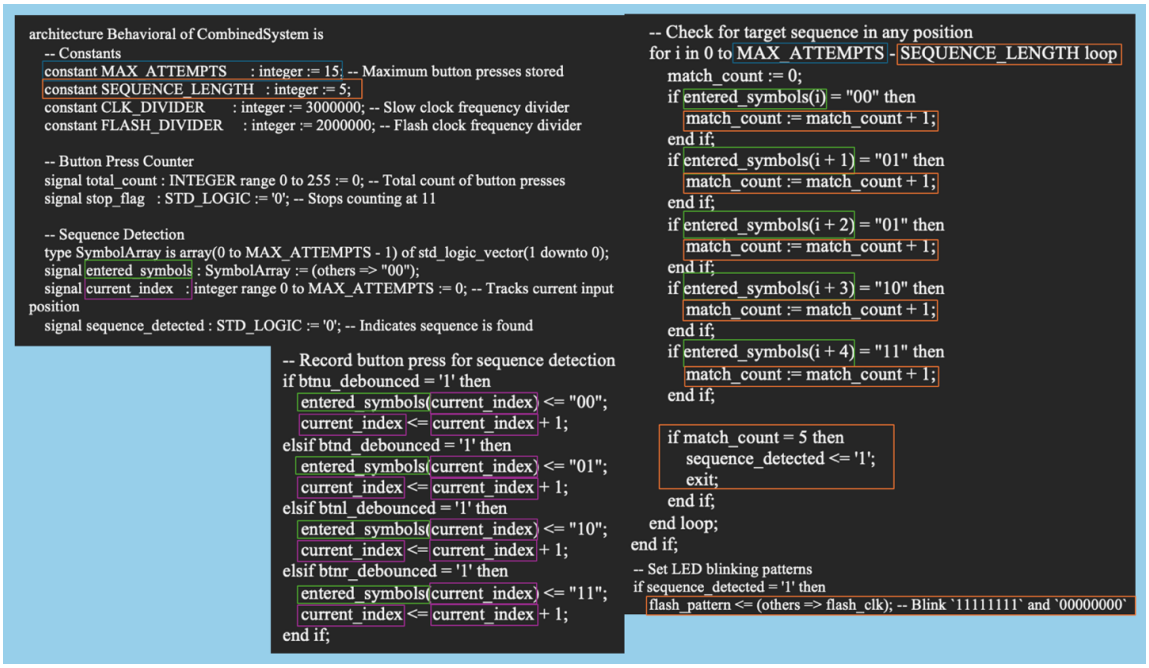
Figure 11 Code figure debounce

Sequence detection state

The sequence detection is used through the code and its critical implementations are to store button symbols, analysing the sequence and triggering actions based on detection. Interestingly, the binary encoding is implemented to stores up to button press symbols and it is declared or wired as current index, which set the range form 0 to max attempts of 15 as constant. As for the logic of the record buttons for sequence detection, the logic is declaring as if else to check if there is an input ''1'' if so, the current index got updated as shown in below code snap figure.

Noticeably, the same binary encoding is used altogether with loop declaration with max attempts (15 encoded symbol but only 4 is used and declared) that the symbol assigned minus sequence length (5 correct sequence) enable the initiation of sequence detection. Interestingly, the simple if and end if logic is used to check the declares (00,01,01,10,11 – btnu, btnd, btnd, btnl, btnr) if the specific input is detected the match count will count and when it is become 5 it provides input ''1'' into sequence detection and it start flashing as shown in below figure.

Furthermore, the entered symbols that use arrays to encode input cleaned debounced buttons is declares and used thought the code as shown in below figure. Noticeably, the enter symbol server as a connection point between record button press and check target sequence in any positions by using enter symbol in checking sequence with current index updates as show in below figure. Interestingly, the below figure 12 table prove the logic behind working of correct sequence logic.



```
architecture Behavioral of CombinedSystem is
    -- Constants
    constant MAX_ATTEMPTS     : integer := 15; -- Maximum button presses stored
    constant SEQUENCE_LENGTH  : integer := 5;
    constant CLK_DIVIDER      : integer := 3000000; -- Slow clock frequency divider
    constant FLASH_DIVIDER    : integer := 2000000; -- Flash clock frequency divider

    -- Button Press Counter
    signal total_count : INTEGER range 0 to 255 := 0; -- Total count of button presses
    signal stop_flag   : STD_LOGIC := '0'; -- Stops counting at 11

    -- Sequence Detection
    type SymbolArray is array(0 to MAX_ATTEMPTS - 1) of std_logic_vector(1 downto 0);
    signal entered_symbols : SymbolArray := (others => "00");
    signal current_index   : integer range 0 to MAX_ATTEMPTS := 0; -- Tracks current input
position
    signal sequence_detected : STD_LOGIC := '0'; -- Indicates sequence is found
```

```
    -- Record button press for sequence detection
    if btnu_debounced = '1' then
        entered_symbols(current_index) <= "00";
        current_index <= current_index + 1;
    elsif btnd_debounced = '1' then
        entered_symbols(current_index) <= "01";
        current_index <= current_index + 1;
    elsif btnl_debounced = '1' then
        entered_symbols(current_index) <= "10";
        current_index <= current_index + 1;
    elsif btnr_debounced = '1' then
        entered_symbols(current_index) <= "11";
        current_index <= current_index + 1;
    end if;
```

```
    -- Check for target sequence in any position
    for i in 0 to MAX_ATTEMPTS - SEQUENCE_LENGTH loop
        match_count := 0;
        if entered_symbols(i) = "00" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 1) = "01" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 2) = "01" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 3) = "10" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 4) = "11" then
            match_count := match_count + 1;
        end if;

        if match_count = 5 then
            sequence_detected <= '1';
            exit;
        end if;
    end loop;
end if;
-- Set LED blinking patterns
if sequence_detected = '1' then
    flash_pattern <= (others => flash_clk); -- Blink `11111111` and `00000000`
```

Position (i)	Expected Symbol	Binary Value	Source Button	Condition for Match
i	"00"	00	btnu	entered_symbols(i) = "00"
i + 1	"01"	01	btnd	entered_symbols(i + 1) = "01"
i + 2	"01"	01	btnd	entered_symbols(i + 2) = "01"
i + 3	"10"	10	btnl	entered_symbols(i + 3) = "10"
i + 4	"11"	11	btnr	entered_symbols(i + 4) = "11"

Figure 12 Correct sequence logic

Lock mechanism

Overall, the main functionality of lock mechanism is to detect sequence, stop flag and btnc reset. Altogether with correct state detection, the lock mechanism is implemented by using the same entered symbols and declaring the stop flag signal when the total count is 11 will provide input ''1'' to stop flag, which further activates lock and the flash lock on, off to off on pattern with the flash signal as shown in below figure 13. The detail explanation of the lock state can be further found in below table in the figure13. Interestingly, the std logic is used to declare an input I or 0 of the flash for further responses.

State	Trigger Condition	Key Signal	LED Pattern	Description
Locked (Default)	Initial state or after pressing btnc.	sequence_detected = 0 stop_flag = 0	Binary count (total_count).	The system is locked. The LEDs display the total count of button presses in binary.
Sequence Unlock	Correct sequence entered: "00", "01", "01", "10", "11".	sequence_detected = 1	Blinking: 11111111 ↔ 00000000	The correct sequence is detected. The system unlocks, and the LEDs blink alternately.
Soft Lock	Button press count reaches 11.	stop_flag = 1	Alternating: 10101010 ↔ 01010101	The system enters a soft lock due to exceeding the button press limit. Input recording stops.
Reset State	Reset button (btnc) pressed.	All signals reset	00000000	Resets the system to its locked state. The LED output is cleared, and the system awaits new inputs.

Figure 13 Lock mechanism,

Counter logic

As for the counter logic, the 16 Hz slow clock used on the rising edge and the noise free clean debounce buttons are used. Among them, the central button is set as a trigger condition '1' with the count reset to zero with stop flag, current index, enter symbol and sequence detection. Furthermore, the code is implemented with elsif sequence detected is zero follow by stop flag as zero to ensure the lock flag display only happen at the total count become eleven as shown in below figure 14.

```
-- Counter and Sequence Detection Process
process(slow_clk)
    variable match_count : integer := 0;  -- Tracks matching symbols
begin
    if rising_edge(slow_clk) then
        if btnc_debounced = '1' then
            -- Reset system
            total_count <= 0;
            stop_flag <= '0';
            current_index <= 0;
            entered_symbols <= (others => "00");
            sequence_detected <= '0';
            led_output <= (others => '0');
        elsif sequence_detected = '0' then
            -- Count button presses
            if stop_flag = '0' then
                if btnu_debounced = '1' or btnd_debounced = '1' or btnl_debounced = '1' or
btnr_debounced = '1' then
                    total_count <= total_count + 1;
                end if;

                -- Stop counting at 11
                if total_count = 11 then
                    stop_flag <= '1';
                end if;
            end if;
```

Figure 14 Counter logic

Difficulties overcome

In first attempt, the overall design was to use finite state machine. Upon carefully consideration, the use of conventional state machine will complicate the design process, thus, the conventional counter that increment from 0 to 255 with stop flag 11 is used to bypass the hardship. As for sequence detection, applying the conventional state assign will require six various conditions to be implemented on code level. However, the use of symbol binary encoding with match count with variable I usage in loop with match count increment when the specific sequence is detected to trigger flash solve the sequence detection issues. Generally, the introducing of buttons and counter interaction introduce newfound bouncing problems. Although there are two potential ways to implement, the three-stage shift register is used in final implementation because of its stable count which remain at input state. Furthermore, the initial design has a problem of implementation flash for both conditions and the use of separate flash clock with well-tuned stable 0.2 s blinking LEDs in the final Implementation become a remedy for best flash application.

System development implementation results analysis

Overall, the three various systems implementations are developed which are the final stable three stage delay bouncing system, the system with bouncing threshold delay and the initial implementation. The detail code for the final stable system can be found in the final code of Appendix 1, while the Appendix 2 covers the initial implementation and the detail code for bouncing with threshold with delay can be found in Flash implementation section of Appendix 1.

Among the three implementations, the final system with stable debounce in below figure is chosen because of its special use of shift register to filter out the noise signal and implementation it will make the input buttons remain on the one state while the other two implementation fail on the inclusion task by having the counting sequence when the button is pressed over the long period.

Also, the other two implementations fail the robustness tests analysis by having delayed debounce threshold with long delay time, thus, the system did not meet the fast-operating buttons for emergency situations as shown in figures 15 below.

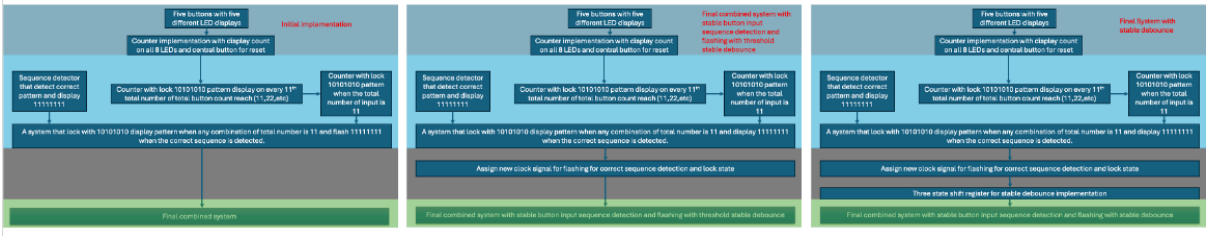


Figure 15 three systems compare

Interestingly, the final stable is operating at 16.67Hz which is higher than the 3.33Hz of the other two, thus, it can accomplish more efficiently with faster response. As for the flash clock adjustment, the same 5Hz with 0.2s response time and 25 cycles per sec clock cycle is set for right blinking rate for elderly with vision problems and distinct flash and on off stop fag display. Furthermore, the debounce threshold for two initial designs is set to 100Hz frequency with 100 cycles per sec operating time for delay for slower response, while the final stable shift registers 3 stage delay use the original slow clock to operate, hence, the synchronization is further achieved. The figure 16 below illustrates the detail calculations.

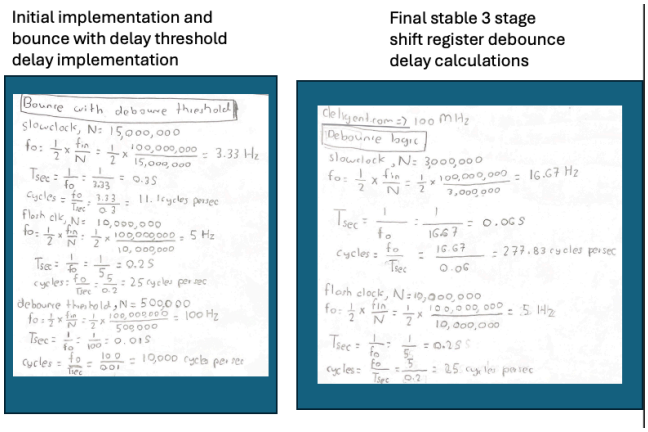


Figure 16 clock freq operating cycle

Resource and Power Utilization

The resource and power utilization analysis of the FPGA-based lock system highlights the efficiency and consistency across all three design iterations. The static power consumption remains at 0.134 W, accounting for 98% of total power usage, while the dynamic power is a minimal 2% or 0.002 W, reflecting the system's efficiency in handling switching and processing tasks. The clock divider, as the core timing and synchronization component, consumes most of the power at 69%, ensuring precise timing for debounce mechanisms, finite state machines (FSMs), and sequence detection. Signal routing and logical elements each contribute 15% to the power usage, emphasizing the efficiency of interconnections and implemented logic across modules. The I/O implementation, responsible for handling the five input buttons and LED outputs, accounts for only 4% of total power, demonstrating its efficiency in supporting reliable user interaction.

Among the three designs, the initial iteration lacked a debounce mechanism and struggled with instability due to switch bounce, leading to inconsistent performance. The second design introduced a threshold delay debounce mechanism, which improved stability but exhibited slower response times, making it less suitable for high-speed or emergency scenarios. The final design employed a three-stage shift register debounce mechanism, offering superior stability and faster response times. Operating at 16.67 Hz, compared to 3.33 Hz in earlier designs, the final iteration ensured robust and efficient operation for critical applications. Furthermore, the final design maintained consistent resource utilization despite incremental improvements, with parallel processing and synchronized clocks enhancing power efficiency.

Overall, the FPGA-based lock system demonstrates a balance between power consumption and functionality. By optimizing key components like the clock divider and debounce mechanism, the final design achieves high performance without exceeding resource constraints. This scalability and energy efficiency make the system ideal for practical applications, especially in scenarios where both responsiveness and power efficiency are critical.

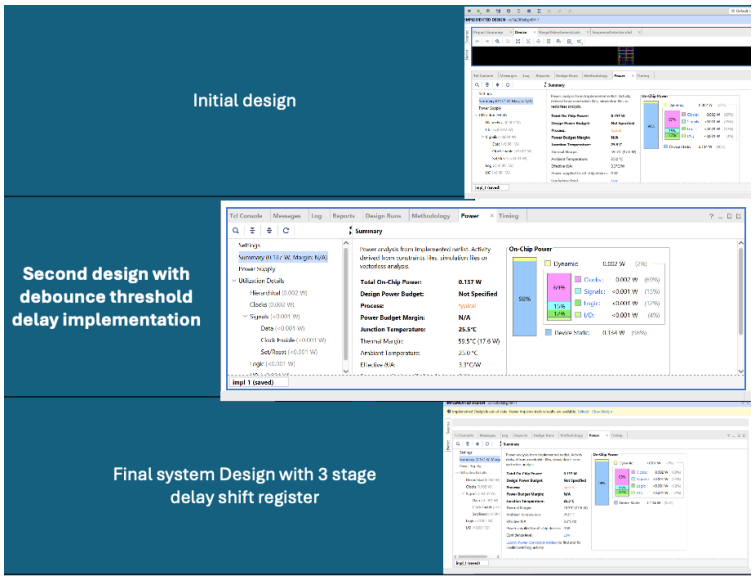


Figure 17 Power utilization

Conclusion

In conclusion, the FPGA-based lock system successfully combines advanced digital logic techniques with a focus on inclusivity and adaptability. The use of a three-stage shift register for debouncing ensures a noise-free and stable operation, making it suitable for environments where precision and speed are critical. The clock divider and FSM-based counter enable efficient timing and sequence detection, while the visual feedback through flashing LEDs ensures intuitive operation. By addressing key challenges such as switch bounce, sequence detection, and resource optimization, the system demonstrates robustness and reliability.

The design effectively meets the needs of diverse user groups, including the elderly and those in high-stress environments, through its responsive and accessible features. However, future iterations could focus on further inclusivity by incorporating features tailored for individuals with disabilities, such as IoT connectivity and custom mechanical switch designs. Overall, this project showcases the potential of FPGA-based systems to deliver efficient, reliable, and user-friendly solutions for modern digital applications.

References

Brown, A. (2018). *Nexys A7 Reference Manual - Digilent Reference*. [online] Digilent.com. Available at: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual?srsltid=AfmBOoquoOU8zAPaC-8mzh1G5dQZRpUHUtY7hIEPVNfJjbpcL0lqNL2r [Accessed 11 Dec. 2024].

Wikipedia. (2020). *Frequency divider*. [online] Available at: https://en.wikipedia.org/wiki/Frequency_divider.

Nuvation Engineering. (2016). *Switch Debouncing for Electronic Product Designs*. [online] Available at: https://www.nuvation.com/resources/article/switch-debouncing-electronic-product-designs?utm_source=chatgpt.com.

The Calculator Site. (n.d.). *Hertz to Seconds Converter*. [online] Available at: https://www.thecalculatorsite.com/conversions/common/hertz-seconds.php.

Li, L., Meng, L. and Wang, F. (2021). Design and simulation of frequency divider circuit based on multisim. *Springer Link (Chiba Institute of Technology)*, 268, pp.01058–01058. doi:https://doi.org/10.1051/e3sconf/202126801058.

Wikipedia Contributors (2024). *Shift register*. Wikipedia.

John (2020). *VHDL Logical Operators and Signal Assignments for Combinational Logic*. [online] FPGA Tutorial. Available at: https://fpgatutorial.com/vhdl-logical-operators-and-signal-assignments-for-combinatorial-logic/.

How To Set a Status Flag in One Clock Domain, Clear It in Another, and Never, Ever Have to Use an Asynchronous Clear for Anything but Reset. (2000). Available at: https://www.floobydust.com/flancter/Flancter_App_Note.pdf [Accessed 11 Dec. 2024].

GeeksforGeeks. (2023). *Serial In Serial Out (SISO) Shift Register*. [online] Available at: https://www.geeksforgeeks.org/siso-shift-register/.

Basic Electronics Tutorials. (2013). *Frequency Division using Divide-by-2 Toggle Flip-flops*. [online] Available at: https://www.electronics-tutorials.ws/counter/count_1.html.

Appendix

Appendix 1

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!Five buttons with five displays!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter is
    Port (
        clk    : in STD_LOGIC;              -- High-frequency clock signal (e.g., 100 MHz)
        btnu   : in STD_LOGIC;               -- Button-up signal
        btnd   : in STD_LOGIC;               -- Button-down signal
        btnl   : in STD_LOGIC;              -- Button-left signal
        btnr   : in STD_LOGIC;              -- Button-right signal
        btnc   : in STD_LOGIC;               -- Button-center signal
        led    : out STD_LOGIC_VECTOR(7 downto 0)  -- 8-bit LED display
    );
end Counter;

architecture Behavioral of Counter is
    signal clk_div       : STD_LOGIC := '0';  -- Slow clock signal for toggling
    signal clk_div_count : integer := 0;      -- Counter for clock division
    constant DIVISOR     : integer := 100000; -- Divide factor for clock (adjust for desired
speed)
    signal pattern       : STD_LOGIC_VECTOR(7 downto 0) := "00000000"; -- LED pattern to
display
begin

    -- Clock Divider Process
    process(clk)
    begin
        if rising_edge(clk) then
            -- Divide the clock signal by the divisor
            if clk_div_count = DIVISOR - 1 then
                clk_div <= not clk_div;  -- Toggle the clock signal
                clk_div_count <= 0;      -- Reset the counter
            else
                clk_div_count <= clk_div_count + 1;  -- Increment the clock divider counter
            end if;
        end if;
    end process;

    -- LED Control Process
    process(btnu, btnd, btnl, btnr, btnc, clk_div)
    begin
        if btnu = '1' then
            -- Button Up: Pattern 1 (Alternating LEDs)
            pattern <= "10101010";
        elsif btnd = '1' then
```

```
            -- Button Down: Pattern 2 (Inverse Alternating LEDs)
            pattern <= "01010101";
        elsif btnl = '1' then
            -- Button Left: Pattern 3 (Outer LEDs)
            pattern <= "10000001";
        elsif btnr = '1' then
            -- Button Right: Pattern 4 (Inner LEDs)
            pattern <= "01111110";
        elsif btnc = '1' then
            -- Button Center: Pattern 5 (All LEDs ON)
            pattern <= "11111111";
        else
            -- No button pressed: Turn off LEDs
            pattern <= "00000000";
        end if;
    end process;

    -- Assign the current pattern to the LEDs
    led <= pattern;

end Behavioral;
```

!!!!!!!Counter that count how many total number of btn up down left right count!!!!!!!!!!!!!!!

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Counterc is
    Port (
        clk    : in STD_LOGIC;               -- Clock signal
        btnu   : in STD_LOGIC;                -- Button Up
        btnd   : in STD_LOGIC;                -- Button Down
        btnl   : in STD_LOGIC;               -- Button Left
        btnr   : in STD_LOGIC;                -- Button Right
        btnc   : in STD_LOGIC;                -- Reset Button
        led    : out STD_LOGIC_VECTOR(7 downto 0)  -- 8-bit LED output
    );
end Counterc;

architecture Behavioral of Counterc is
    -- Button Press Counter
    signal total_count : INTEGER range 0 to 255 := 0; -- Total count of button presses

    -- Debounce Signals
    signal btnu_stable, btnd_stable, btnl_stable, btnr_stable : STD_LOGIC := '0';
    signal btnu_debounce, btnd_debounce, btnl_debounce, btnr_debounce :
INTEGER range 0 to 500000 := 0;
    constant DEBOUNCE_THRESHOLD : INTEGER := 500000; -- Adjust for
debounce timing
```

```
    -- Clock Divider for LED Updates
    signal slow_clk : STD_LOGIC := '0';
    signal clk_count : INTEGER := 0;
    constant CLK_DIVIDER : INTEGER := 500000; -- Adjust for slower clock
begin

    -- Clock Divider Process
    process(clk)
    begin
        if rising_edge(clk) then
            if clk_count = CLK_DIVIDER then
                slow_clk <= not slow_clk; -- Toggle slow clock
                clk_count <= 0;
            else
                clk_count <= clk_count + 1;
            end if;
        end if;
    end process;

    -- Debounce Process
    process(clk)
    begin
        if rising_edge(clk) then
            -- Debounce btnu
            if btnu = '1' then
                if btnu_debounce < DEBOUNCE_THRESHOLD then
                    btnu_debounce <= btnu_debounce + 1;
                else
                    btnu_stable <= '1';
                end if;
            else
                btnu_debounce <= 0;
                btnu_stable <= '0';
            end if;

            -- Debounce btnd
            if btnd = '1' then
                if btnd_debounce < DEBOUNCE_THRESHOLD then
                    btnd_debounce <= btnd_debounce + 1;
                else
                    btnd_stable <= '1';
                end if;
            else
                btnd_debounce <= 0;
                btnd_stable <= '0';
            end if;

            -- Debounce btnl
            if btnl = '1' then
```

```
            if btnl_debounce < DEBOUNCE_THRESHOLD then
                btnl_debounce <= btnl_debounce + 1;
            else
                btnl_stable <= '1';
            end if;
        else
            btnl_debounce <= 0;
            btnl_stable <= '0';
        end if;

        -- Debounce btnr
        if btnr = '1' then
            if btnr_debounce < DEBOUNCE_THRESHOLD then
                btnr_debounce <= btnr_debounce + 1;
            else
                btnr_stable <= '1';
            end if;
        else
            btnr_debounce <= 0;
            btnr_stable <= '0';
        end if;
    end if;
end process;

-- Counting Button Presses
process(slow_clk)
begin
    if rising_edge(slow_clk) then
        if btnc = '1' then
            -- Reset the counter
            total_count <= 0;
        else
            -- Increment the counter for each stable button press
            if btnu_stable = '1' or btnd_stable = '1' or btnl_stable = '1' or btnr_stable =
'1' then
                total_count <= total_count + 1;
            end if;
        end if;
    end if;
end process;

-- LED Output
process(slow_clk)
begin
    if rising_edge(slow_clk) then
        -- Display total_count on LEDs
        led <= std_logic_vector(to_unsigned(total_count, 8));
    end if;
end process;
```

```
end
Behavioral;

!!!!!!!!!!!!!!!!!!!!!!!!GENERATE 10101010 ON EVERY 11 TH BOTTOM
PRESS!!!!!!!!!!!!!!!!!!!!!!
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Counterc is
    Port (
        clk    : in STD_LOGIC;              -- Clock signal
        btnu   : in STD_LOGIC;               -- Button Up
        btnd   : in STD_LOGIC;               -- Button Down
        btnl   : in STD_LOGIC;              -- Button Left
        btnr   : in STD_LOGIC;              -- Button Right
        btnc   : in STD_LOGIC;               -- Reset Button
        led    : out STD_LOGIC_VECTOR(7 downto 0)  -- 8-bit LED output
    );
end Counterc;

architecture Behavioral of Counterc is
    -- Button Press Counter
    signal total_count : INTEGER range 0 to 255 := 0; -- Total count of button presses

    -- Debounce Signals
    signal btnu_stable, btnd_stable, btnl_stable, btnr_stable : STD_LOGIC := '0';
    signal btnu_debounce, btnd_debounce, btnl_debounce, btnr_debounce : INTEGER range 0
to 500000 := 0;
    constant DEBOUNCE_THRESHOLD : INTEGER := 500000; -- Adjust for debounce
timing

    -- Clock Divider for LED Updates
    signal slow_clk : STD_LOGIC := '0';
    signal clk_count : INTEGER := 0;
    constant CLK_DIVIDER : INTEGER := 500000; -- Adjust for slower clock
begin

    -- Clock Divider Process
    process(clk)
    begin
        if rising_edge(clk) then
            if clk_count = CLK_DIVIDER then
                slow_clk <= not slow_clk; -- Toggle slow clock
                clk_count <= 0;
            else
                clk_count <= clk_count + 1;
            end if;
        end if;
    end process;
```

```
-- Debounce Process
process(clk)
begin
   if rising_edge(clk) then
      -- Debounce btnu
      if btnu = '1' then
         if btnu_debounce < DEBOUNCE_THRESHOLD then
            btnu_debounce <= btnu_debounce + 1;
         else
            btnu_stable <= '1';
         end if;
      else
         btnu_debounce <= 0;
         btnu_stable <= '0';
      end if;

      -- Debounce btnd
      if btnd = '1' then
         if btnd_debounce < DEBOUNCE_THRESHOLD then
            btnd_debounce <= btnd_debounce + 1;
         else
            btnd_stable <= '1';
         end if;
      else
         btnd_debounce <= 0;
         btnd_stable <= '0';
      end if;

      -- Debounce btnl
      if btnl = '1' then
         if btnl_debounce < DEBOUNCE_THRESHOLD then
            btnl_debounce <= btnl_debounce + 1;
         else
            btnl_stable <= '1';
         end if;
      else
         btnl_debounce <= 0;
         btnl_stable <= '0';
      end if;

      -- Debounce btnr
      if btnr = '1' then
         if btnr_debounce < DEBOUNCE_THRESHOLD then
            btnr_debounce <= btnr_debounce + 1;
         else
            btnr_stable <= '1';
         end if;
      else
         btnr_debounce <= 0;
```

```
            btnr_stable <= '0';
        end if;
    end if;
end process;

-- Counting Button Presses
process(slow_clk)
begin
    if rising_edge(slow_clk) then
        if btnc = '1' then
            -- Reset the counter
            total_count <= 0;
        else
            -- Increment the counter for each stable button press
            if btnu_stable = '1' or btnd_stable = '1' or btnl_stable = '1' or btnr_stable = '1' then
                total_count <= total_count + 1;
            end if;
        end if;
    end if;
end process;

-- LED Output
process(slow_clk)
begin
    if rising_edge(slow_clk) then
        if btnc = '1' then
            -- Clear LEDs on reset
            led <= (others => '0');
        elsif total_count mod 11 = 0 and total_count /= 0 then
            -- Display 10101010 on every 11th button press
            led <= "10101010";
        else
            -- Display total_count on LEDs
            led <= std_logic_vector(to_unsigned(total_count, 8));
        end if;
    end if;
end process;
end Behavioral;
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!lock the system when 11 is inputted!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Counterc is
    Port (
        clk    : in STD_LOGIC;              -- Clock signal
        btnu   : in STD_LOGIC;               -- Button Up
        btnd   : in STD_LOGIC;               -- Button Down
        btnl   : in STD_LOGIC;              -- Button Left
        btnr   : in STD_LOGIC;               -- Button Right
```

```
        btnc   : in STD_LOGIC;               -- Reset Button
        led    : out STD_LOGIC_VECTOR(7 downto 0)  -- 8-bit LED output
    );
end Counterc;

architecture Behavioral of Counterc is
    -- Button Press Counter
    signal total_count : INTEGER range 0 to 255 := 0; -- Total count of button presses

    -- Debounce Signals
    signal btnu_stable, btnd_stable, btnl_stable, btnr_stable : STD_LOGIC := '0';
    signal btnu_debounce, btnd_debounce, btnl_debounce, btnr_debounce : INTEGER range 0
to 500000 := 0;
    constant DEBOUNCE_THRESHOLD : INTEGER := 500000; -- Adjust for debounce
timing

    -- Clock Divider for LED Updates
    signal slow_clk : STD_LOGIC := '0';
    signal clk_count : INTEGER := 0;
    constant CLK_DIVIDER : INTEGER := 21_000_000; -- Slightly increased for slower
operation
    signal stop_flag : STD_LOGIC := '0'; -- Flag to stop the system
begin

    -- Clock Divider Process
    process(clk)
    begin
        if rising_edge(clk) then
            if clk_count = CLK_DIVIDER then
                slow_clk <= not slow_clk; -- Toggle slow clock
                clk_count <= 0;
            else
                clk_count <= clk_count + 1;
            end if;
        end if;
    end process;

    -- Debounce Process
    process(clk)
    begin
        if rising_edge(clk) then
            -- Debounce btnu
            if btnu = '1' then
                if btnu_debounce < DEBOUNCE_THRESHOLD then
                    btnu_debounce <= btnu_debounce + 1;
                else
                    btnu_stable <= '1';
                end if;
            else
                btnu_debounce <= 0;
```

```
            btnu_stable <= '0';
        end if;

        -- Debounce btnd
        if btnd = '1' then
            if btnd_debounce < DEBOUNCE_THRESHOLD then
                btnd_debounce <= btnd_debounce + 1;
            else
                btnd_stable <= '1';
            end if;
        else
            btnd_debounce <= 0;
            btnd_stable <= '0';
        end if;

        -- Debounce btnl
        if btnl = '1' then
            if btnl_debounce < DEBOUNCE_THRESHOLD then
                btnl_debounce <= btnl_debounce + 1;
            else
                btnl_stable <= '1';
            end if;
        else
            btnl_debounce <= 0;
            btnl_stable <= '0';
        end if;

        -- Debounce btnr
        if btnr = '1' then
            if btnr_debounce < DEBOUNCE_THRESHOLD then
                btnr_debounce <= btnr_debounce + 1;
            else
                btnr_stable <= '1';
            end if;
        else
            btnr_debounce <= 0;
            btnr_stable <= '0';
        end if;
    end if;
end process;

-- Counting Button Presses
process(slow_clk)
begin
    if rising_edge(slow_clk) then
        if btnc = '1' then
            -- Reset the counter and clear the stop flag
            total_count <= 0;
            stop_flag <= '0';
        elsif stop_flag = '0' then
```

```
            -- Increment the counter for each stable button press if not stopped
            if btnu_stable = '1' or btnd_stable = '1' or btnl_stable = '1' or btnr_stable = '1' then
                total_count <= total_count + 1;
            end if;

            -- Stop when total_count reaches 11
            if total_count = 11 then
                stop_flag <= '1'; -- Activate the stop flag
            end if;
        end if;
    end if;
end process;

-- LED Output
process(slow_clk)
begin
    if rising_edge(slow_clk) then
        if btnc = '1' then
            -- Clear LEDs on reset
            led <= (others => '0');
        elsif stop_flag = '1' then
            -- Freeze LEDs to 10101010 when stopped
            led <= "10101010";
        else
            -- Display total_count on LEDs
            led <= std_logic_vector(to_unsigned(total_count, 8));
        end if;
    end if;
end process;

end Behavioral;

!!!!!!!!!!!!!!!!!!!!!!Sequence detector!detect pattern up,down,down,left,right!!!!!!!!!!!!!!!!!!!!
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SequenceDetector is
    Port (
        clk     : in  STD_LOGIC;              -- Clock signal
        btnc    : in  STD_LOGIC;              -- Initialization button (center)
        btnu    : in  STD_LOGIC;              -- Button Up
        btnd    : in  STD_LOGIC;              -- Button Down
        btnl    : in  STD_LOGIC;              -- Button Left
        btnr    : in  STD_LOGIC;              -- Button Right
        led     : out STD_LOGIC_VECTOR(7 downto 0) -- LED output
    );
end SequenceDetector;

architecture Behavioral of SequenceDetector is
```

```
    -- Constants
    constant MAX_ATTEMPTS : integer := 15; -- Maximum button presses stored
    constant SEQUENCE_LENGTH : integer := 5;

    -- Signal declarations
    type SymbolArray is array(0 to MAX_ATTEMPTS - 1) of std_logic_vector(1 downto 0);
    signal entered_symbols : SymbolArray := (others => "00");
    signal current_index : integer range 0 to MAX_ATTEMPTS := 0; -- Tracks current input
position
    signal sequence_detected : STD_LOGIC := '0'; -- Indicates sequence is found
    signal led_output : STD_LOGIC_VECTOR(7 downto 0) := (others => '0'); -- LED output

    -- Button debounce signals
    signal btnu_stable, btnd_stable, btnl_stable, btnr_stable, btnc_stable : STD_LOGIC := '0';
    signal btnu_debounce, btnd_debounce, btnl_debounce, btnr_debounce, btnc_debounce :
integer range 0 to 1000000 := 0;
    constant DEBOUNCE_THRESHOLD : integer := 500000; -- Debounce threshold

    -- Clock divider
    signal slow_clk : STD_LOGIC := '0';
    signal clk_count : integer := 0;
    constant CLK_DIVIDER : integer := 20000000; -- Slower clock for processing
begin
    -- Clock divider process
    process(clk)
    begin
        if rising_edge(clk) then
            if clk_count = CLK_DIVIDER then
                slow_clk <= not slow_clk;
                clk_count <= 0;
            else
                clk_count <= clk_count + 1;
            end if;
        end if;
    end process;

    -- Debounce process for stable button signals
    process(clk)
    begin
        if rising_edge(clk) then
            -- Debounce logic for each button
            if btnu = '1' then
                if btnu_debounce < DEBOUNCE_THRESHOLD then
                    btnu_debounce <= btnu_debounce + 1;
                else
                    btnu_stable <= '1';
                end if;
            else
                btnu_debounce <= 0;
                btnu_stable <= '0';
```

```
        end if;

        if btnd = '1' then
            if btnd_debounce < DEBOUNCE_THRESHOLD then
                btnd_debounce <= btnd_debounce + 1;
            else
                btnd_stable <= '1';
            end if;
        else
            btnd_debounce <= 0;
            btnd_stable <= '0';
        end if;

        if btnl = '1' then
            if btnl_debounce < DEBOUNCE_THRESHOLD then
                btnl_debounce <= btnl_debounce + 1;
            else
                btnl_stable <= '1';
            end if;
        else
            btnl_debounce <= 0;
            btnl_stable <= '0';
        end if;

        if btnr = '1' then
            if btnr_debounce < DEBOUNCE_THRESHOLD then
                btnr_debounce <= btnr_debounce + 1;
            else
                btnr_stable <= '1';
            end if;
        else
            btnr_debounce <= 0;
            btnr_stable <= '0';
        end if;

        if btnc = '1' then
            if btnc_debounce < DEBOUNCE_THRESHOLD then
                btnc_debounce <= btnc_debounce + 1;
            else
                btnc_stable <= '1';
            end if;
        else
            btnc_debounce <= 0;
            btnc_stable <= '0';
        end if;
    end if;
end process;

-- Sequence detection and LED output process
process(slow_clk)
```

```
    variable temp_sequence : SymbolArray; -- Temporary sequence holder
  begin
    if rising_edge(slow_clk) then
      if btnc_stable = '1' then
        -- Reset system
        current_index <= 0;
        entered_symbols <= (others => "00");
        sequence_detected <= '0';
        led_output <= (others => '0');
      elsif current_index < MAX_ATTEMPTS then
        -- Record button press
        if btnu_stable = '1' then
          entered_symbols(current_index) <= "00";
          current_index <= current_index + 1;
        elsif btnd_stable = '1' then
          entered_symbols(current_index) <= "01";
          current_index <= current_index + 1;
        elsif btnl_stable = '1' then
          entered_symbols(current_index) <= "10";
          current_index <= current_index + 1;
        elsif btnr_stable = '1' then
          entered_symbols(current_index) <= "11";
          current_index <= current_index + 1;
        end if;

        -- Check for the specific sequences
        temp_sequence := entered_symbols;
        if (temp_sequence(0) = "00" and temp_sequence(1) = "01" and
          temp_sequence(2) = "01" and temp_sequence(3) = "10" and
          temp_sequence(4) = "11") then
          sequence_detected <= '1';
        elsif (temp_sequence(1) = "00" and temp_sequence(2) = "01" and
            temp_sequence(3) = "01" and temp_sequence(4) = "10" and
            temp_sequence(5) = "11") then
          sequence_detected <= '1';
        end if;
      end if;

      -- Set LED output based on sequence detection
      if sequence_detected = '1' then
        led_output <= (others => '1'); -- Display `11111111`
      else
        led_output <= std_logic_vector(to_unsigned(current_index, 8)); -- Default to
current index
      end if;
    end if;
  end process;

  -- Assign LED output
  led <= led_output;
```

end Behavioral;

!!!Combined System!!

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CombinedSystem is
   Port (
      clk    : in  STD_LOGIC;              -- Clock signal
      btnc   : in  STD_LOGIC;              -- Reset button
      btnu   : in  STD_LOGIC;              -- Button Up
      btnd   : in  STD_LOGIC;              -- Button Down
      btnl   : in  STD_LOGIC;              -- Button Left
      btnr   : in  STD_LOGIC;              -- Button Right
      led    : out STD_LOGIC_VECTOR(7 downto 0) -- LED output
   );
end CombinedSystem;

architecture Behavioral of CombinedSystem is
   -- Constants
   constant MAX_ATTEMPTS : integer := 15; -- Maximum button presses stored
   constant SEQUENCE_LENGTH : integer := 5;
   constant DEBOUNCE_THRESHOLD : integer := 500000; -- Debounce threshold
   constant CLK_DIVIDER : integer := 15000000; -- Reduced Clock divider for faster
processing

   -- Button Press Counter
   signal total_count : INTEGER range 0 to 255 := 0; -- Total count of button presses
   signal stop_flag : STD_LOGIC := '0'; -- Stops counting at 11

   -- Sequence Detection
   type SymbolArray is array(0 to MAX_ATTEMPTS - 1) of std_logic_vector(1 downto 0);
   signal entered_symbols : SymbolArray := (others => "00");
   signal current_index : integer range 0 to MAX_ATTEMPTS := 0; -- Tracks current input
position
   signal sequence_detected : STD_LOGIC := '0'; -- Indicates sequence is found

   -- Clock Divider
   signal slow_clk : STD_LOGIC := '0';
   signal clk_count : integer := 0;

   -- Debounce Signals
   signal btnu_stable, btnd_stable, btnl_stable, btnr_stable, btnc_stable : STD_LOGIC := '0';
   signal btnu_debounce, btnd_debounce, btnl_debounce, btnr_debounce, btnc_debounce :
integer range 0 to 1000000 := 0;

   -- LED Output
   signal led_output : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
```

```
begin
    -- Clock Divider Process
    process(clk)
    begin
        if rising_edge(clk) then
            if clk_count = CLK_DIVIDER then
                slow_clk <= not slow_clk;
                clk_count <= 0;
            else
                clk_count <= clk_count + 1;
            end if;
        end if;
    end process;

    -- Debounce Process
    process(clk)
    begin
        if rising_edge(clk) then
            -- Debounce btnu
            if btnu = '1' then
                if btnu_debounce < DEBOUNCE_THRESHOLD then
                    btnu_debounce <= btnu_debounce + 1;
                else
                    btnu_stable <= '1';
                end if;
            else
                btnu_debounce <= 0;
                btnu_stable <= '0';
            end if;

            -- Debounce btnd
            if btnd = '1' then
                if btnd_debounce < DEBOUNCE_THRESHOLD then
                    btnd_debounce <= btnd_debounce + 1;
                else
                    btnd_stable <= '1';
                end if;
            else
                btnd_debounce <= 0;
                btnd_stable <= '0';
            end if;

            -- Debounce btnl
            if btnl = '1' then
                if btnl_debounce < DEBOUNCE_THRESHOLD then
                    btnl_debounce <= btnl_debounce + 1;
                else
                    btnl_stable <= '1';
                end if;
            else
```

```
            btnl_debounce <= 0;
            btnl_stable <= '0';
        end if;

        -- Debounce btnr
        if btnr = '1' then
            if btnr_debounce < DEBOUNCE_THRESHOLD then
                btnr_debounce <= btnr_debounce + 1;
            else
                btnr_stable <= '1';
            end if;
        else
            btnr_debounce <= 0;
            btnr_stable <= '0';
        end if;

        -- Debounce btnc
        if btnc = '1' then
            if btnc_debounce < DEBOUNCE_THRESHOLD then
                btnc_debounce <= btnc_debounce + 1;
            else
                btnc_stable <= '1';
            end if;
        else
            btnc_debounce <= 0;
            btnc_stable <= '0';
        end if;
    end if;
end process;

-- Counter and Sequence Detection Process
process(slow_clk)
    variable temp_sequence : SymbolArray; -- Temporary sequence holder
    variable match_count : integer := 0;  -- Tracks matching symbols
begin
    if rising_edge(slow_clk) then
        if btnc_stable = '1' then
            -- Reset system
            total_count <= 0;
            stop_flag <= '0';
            current_index <= 0;
            entered_symbols <= (others => "00");
            sequence_detected <= '0';
            led_output <= (others => '0');
        elsif sequence_detected = '0' then
            -- Count button presses
            if stop_flag = '0' then
                if btnu_stable = '1' or btnd_stable = '1' or btnl_stable = '1' or btnr_stable = '1' then
                    total_count <= total_count + 1;
                end if;
```

```
        -- Stop counting at 11
        if total_count = 11 then
            stop_flag <= '1';
        end if;
    end if;

    -- Record button press for sequence detection
    if btnu_stable = '1' then
        entered_symbols(current_index) <= "00";
        current_index <= current_index + 1;
    elsif btnd_stable = '1' then
        entered_symbols(current_index) <= "01";
        current_index <= current_index + 1;
    elsif btnl_stable = '1' then
        entered_symbols(current_index) <= "10";
        current_index <= current_index + 1;
    elsif btnr_stable = '1' then
        entered_symbols(current_index) <= "11";
        current_index <= current_index + 1;
    end if;

    -- Check for target sequence in any position
    for i in 0 to MAX_ATTEMPTS - SEQUENCE_LENGTH loop
        match_count := 0;
        if entered_symbols(i) = "00" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 1) = "01" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 2) = "01" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 3) = "10" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 4) = "11" then
            match_count := match_count + 1;
        end if;

        if match_count = 5 then
            sequence_detected <= '1';
            exit;
        end if;
    end loop;
end if;

-- Set LED output
if sequence_detected = '1' then
```

```
            led_output <= (others => '1'); -- Display `11111111`
        elsif stop_flag = '1' then
            led_output <= "10101010"; -- Display `10101010` when stopped
        else
            led_output <= std_logic_vector(to_unsigned(total_count, 8)); -- Default to count
        end if;
    end if;
end process;

-- Assign LED output
led <= led_output;

end Behavioral;
```

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!Flash Implementation!!

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CombinedSystem is
    Port (
        clk    : in  STD_LOGIC;                -- Clock signal
        btnc   : in  STD_LOGIC;                -- Reset button
        btnu   : in  STD_LOGIC;                -- Button Up
        btnd   : in  STD_LOGIC;                -- Button Down
        btnl   : in  STD_LOGIC;                -- Button Left
        btnr   : in  STD_LOGIC;                -- Button Right
        led    : out STD_LOGIC_VECTOR(7 downto 0) -- LED output
    );
end CombinedSystem;

architecture Behavioral of CombinedSystem is
    -- Constants
    constant MAX_ATTEMPTS : integer := 15; -- Maximum button presses stored
    constant SEQUENCE_LENGTH : integer := 5;
    constant DEBOUNCE_THRESHOLD : integer := 500000; -- Debounce threshold
    constant CLK_DIVIDER : integer := 15000000; -- Reduced Clock divider for faster
processing
    constant FLASH_DIVIDER : integer := 10000000; -- Divider for blinking LEDs

    -- Button Press Counter
    signal total_count : INTEGER range 0 to 255 := 0; -- Total count of button presses
    signal stop_flag : STD_LOGIC := '0'; -- Stops counting at 11

    -- Sequence Detection
    type SymbolArray is array(0 to MAX_ATTEMPTS - 1) of std_logic_vector(1 downto 0);
    signal entered_symbols : SymbolArray := (others => "00");
    signal current_index : integer range 0 to MAX_ATTEMPTS := 0; -- Tracks current input
position
```

```
    signal sequence_detected : STD_LOGIC := '0'; -- Indicates sequence is found

    -- Clock Divider
    signal slow_clk : STD_LOGIC := '0';
    signal flash_clk : STD_LOGIC := '0'; -- Blinking clock
    signal clk_count : integer := 0;
    signal flash_count : integer := 0;

    -- Debounce Signals
    signal btnu_stable, btnd_stable, btnl_stable, btnr_stable, btnc_stable : STD_LOGIC := '0';
    signal btnu_debounce, btnd_debounce, btnl_debounce, btnr_debounce, btnc_debounce :
integer range 0 to 1000000 := 0;

    -- LED Output
    signal led_output : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal flash_pattern : STD_LOGIC_VECTOR(7 downto 0) := (others => '0'); -- Temporary
pattern for blinking
begin
    -- Clock Divider Process
    process(clk)
    begin
        if rising_edge(clk) then
            -- Generate slow clock
            if clk_count = CLK_DIVIDER then
                slow_clk <= not slow_clk;
                clk_count <= 0;
            else
                clk_count <= clk_count + 1;
            end if;

            -- Generate flash clock for blinking
            if flash_count = FLASH_DIVIDER then
                flash_clk <= not flash_clk;
                flash_count <= 0;
            else
                flash_count <= flash_count + 1;
            end if;
        end if;
    end process;

    -- Debounce Process
    process(clk)
    begin
        if rising_edge(clk) then
            -- Debounce btnu
            if btnu = '1' then
                if btnu_debounce < DEBOUNCE_THRESHOLD then
                    btnu_debounce <= btnu_debounce + 1;
                else
                    btnu_stable <= '1';
```

```
    end if;
else
    btnu_debounce <= 0;
    btnu_stable <= '0';
end if;

-- Debounce btnd
if btnd = '1' then
    if btnd_debounce < DEBOUNCE_THRESHOLD then
        btnd_debounce <= btnd_debounce + 1;
    else
        btnd_stable <= '1';
    end if;
else
    btnd_debounce <= 0;
    btnd_stable <= '0';
end if;

-- Debounce btnl
if btnl = '1' then
    if btnl_debounce < DEBOUNCE_THRESHOLD then
        btnl_debounce <= btnl_debounce + 1;
    else
        btnl_stable <= '1';
    end if;
else
    btnl_debounce <= 0;
    btnl_stable <= '0';
end if;

-- Debounce btnr
if btnr = '1' then
    if btnr_debounce < DEBOUNCE_THRESHOLD then
        btnr_debounce <= btnr_debounce + 1;
    else
        btnr_stable <= '1';
    end if;
else
    btnr_debounce <= 0;
    btnr_stable <= '0';
end if;

-- Debounce btnc
if btnc = '1' then
    if btnc_debounce < DEBOUNCE_THRESHOLD then
        btnc_debounce <= btnc_debounce + 1;
    else
        btnc_stable <= '1';
    end if;
else
```

```
            btnc_debounce <= 0;
            btnc_stable <= '0';
        end if;
    end if;
end process;

-- Counter and Sequence Detection Process
process(slow_clk)
    variable temp_sequence : SymbolArray; -- Temporary sequence holder
    variable match_count : integer := 0;  -- Tracks matching symbols
begin
    if rising_edge(slow_clk) then
        if btnc_stable = '1' then
            -- Reset system
            total_count <= 0;
            stop_flag <= '0';
            current_index <= 0;
            entered_symbols <= (others => "00");
            sequence_detected <= '0';
            led_output <= (others => '0');
        elsif sequence_detected = '0' then
            -- Count button presses
            if stop_flag = '0' then
                if btnu_stable = '1' or btnd_stable = '1' or btnl_stable = '1' or btnr_stable = '1' then
                    total_count <= total_count + 1;
                end if;

                -- Stop counting at 11
                if total_count = 11 then
                    stop_flag <= '1';
                end if;
            end if;

            -- Record button press for sequence detection
            if btnu_stable = '1' then
                entered_symbols(current_index) <= "00";
                current_index <= current_index + 1;
            elsif btnd_stable = '1' then
                entered_symbols(current_index) <= "01";
                current_index <= current_index + 1;
            elsif btnl_stable = '1' then
                entered_symbols(current_index) <= "10";
                current_index <= current_index + 1;
            elsif btnr_stable = '1' then
                entered_symbols(current_index) <= "11";
                current_index <= current_index + 1;
            end if;

            -- Check for target sequence in any position
            for i in 0 to MAX_ATTEMPTS - SEQUENCE_LENGTH loop
```

```
                match_count := 0;
                if entered_symbols(i) = "00" then
                    match_count := match_count + 1;
                end if;
                if entered_symbols(i + 1) = "01" then
                    match_count := match_count + 1;
                end if;
                if entered_symbols(i + 2) = "01" then
                    match_count := match_count + 1;
                end if;
                if entered_symbols(i + 3) = "10" then
                    match_count := match_count + 1;
                end if;
                if entered_symbols(i + 4) = "11" then
                    match_count := match_count + 1;
                end if;

                if match_count = 5 then
                    sequence_detected <= '1';
                    exit;
                end if;
            end loop;
        end if;

        -- Set LED blinking patterns
        if sequence_detected = '1' then
            flash_pattern <= (others => flash_clk); -- Blink `11111111` and `00000000`
        elsif stop_flag = '1' then
            if flash_clk = '1' then
                flash_pattern <= "10101010";
            else
                flash_pattern <= "01010101";
            end if;
        else
            flash_pattern <= std_logic_vector(to_unsigned(total_count, 8)); -- Default to count
        end if;

        led_output <= flash_pattern;
    end if;
end process;

-- Assign LED output
led <= led_output;

end Behavioral;
```

!!!!!!!!!!!!!!!!!!Stable debounce implementation !!!

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

```
entity CombinedSystem is
    Port (
        clk    : in  STD_LOGIC;              -- Clock signal
        btnc   : in  STD_LOGIC;               -- Reset button
        btnu   : in  STD_LOGIC;               -- Button Up
        btnd   : in  STD_LOGIC;               -- Button Down
        btnl   : in  STD_LOGIC;              -- Button Left
        btnr   : in  STD_LOGIC;              -- Button Right
        led    : out STD_LOGIC_VECTOR(7 downto 0) -- LED output
    );
end CombinedSystem;

architecture Behavioral of CombinedSystem is
    -- Constants
    constant MAX_ATTEMPTS      : integer := 15; -- Maximum button presses stored
    constant SEQUENCE_LENGTH   : integer := 5;
    constant CLK_DIVIDER       : integer := 12000000; -- Slow clock frequency divider
    constant FLASH_DIVIDER     : integer := 8000000; -- Flash clock frequency divider

    -- Button Press Counter
    signal total_count : INTEGER range 0 to 255 := 0; -- Total count of button presses
    signal stop_flag   : STD_LOGIC := '0'; -- Stops counting at 11

    -- Sequence Detection
    type SymbolArray is array(0 to MAX_ATTEMPTS - 1) of std_logic_vector(1 downto 0);
    signal entered_symbols : SymbolArray := (others => "00");
    signal current_index   : integer range 0 to MAX_ATTEMPTS := 0; -- Tracks current input
position
    signal sequence_detected : STD_LOGIC := '0'; -- Indicates sequence is found

    -- Clock Divider
    signal slow_clk  : STD_LOGIC := '0';
    signal flash_clk : STD_LOGIC := '0'; -- Blinking clock
    signal clk_count : integer := 0;
    signal flash_count : integer := 0;

    -- Debouncer Outputs
    signal btnu_debounced, btnd_debounced, btnl_debounced, btnr_debounced,
btnc_debounced : std_logic;

    -- LED Output
    signal led_output    : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal flash_pattern : STD_LOGIC_VECTOR(7 downto 0) := (others => '0'); -- Temporary
pattern for blinking
begin
    -- Clock Divider Process
    process(clk)
    begin
        if rising_edge(clk) then
```

```
        -- Generate slow clock
        if clk_count = CLK_DIVIDER - 1 then
            slow_clk <= not slow_clk;
            clk_count <= 0;
        else
            clk_count <= clk_count + 1;
        end if;

        -- Generate flash clock for blinking
        if flash_count = FLASH_DIVIDER - 1 then
            flash_clk <= not flash_clk;
            flash_count <= 0;
        else
            flash_count <= flash_count + 1;
        end if;
    end if;
end process;

-- Instantiate Debouncer for Each Button
debounce_btnu: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
        input  => btnu,
        output => btnu_debounced
    );

debounce_btnd: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
        input  => btnd,
        output => btnd_debounced
    );

debounce_btnl: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
        input  => btnl,
        output => btnl_debounced
    );

debounce_btnr: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
        input  => btnr,
        output => btnr_debounced
    );

debounce_btnc: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
```

```
        input  => btnc,
        output => btnc_debounced
    );

-- Counter and Sequence Detection Process
process(slow_clk)
    variable match_count : integer := 0;  -- Tracks matching symbols
begin
    if rising_edge(slow_clk) then
        if btnc_debounced = '1' then
            -- Reset system
            total_count <= 0;
            stop_flag <= '0';
            current_index <= 0;
            entered_symbols <= (others => "00");
            sequence_detected <= '0';
            led_output <= (others => '0');
        elsif sequence_detected = '0' then
            -- Count button presses
            if stop_flag = '0' then
                if btnu_debounced = '1' or btnd_debounced = '1' or btnl_debounced = '1' or
btnr_debounced = '1' then
                    total_count <= total_count + 1;
                end if;

                -- Stop counting at 11
                if total_count = 11 then
                    stop_flag <= '1';
                end if;
            end if;

            -- Record button press for sequence detection
            if btnu_debounced = '1' then
                entered_symbols(current_index) <= "00";
                current_index <= current_index + 1;
            elsif btnd_debounced = '1' then
                entered_symbols(current_index) <= "01";
                current_index <= current_index + 1;
            elsif btnl_debounced = '1' then
                entered_symbols(current_index) <= "10";
                current_index <= current_index + 1;
            elsif btnr_debounced = '1' then
                entered_symbols(current_index) <= "11";
                current_index <= current_index + 1;
            end if;

            -- Check for target sequence in any position
            for i in 0 to MAX_ATTEMPTS - SEQUENCE_LENGTH loop
                match_count := 0;
                if entered_symbols(i) = "00" then
```

```
                match_count := match_count + 1;
            end if;
            if entered_symbols(i + 1) = "01" then
                match_count := match_count + 1;
            end if;
            if entered_symbols(i + 2) = "01" then
                match_count := match_count + 1;
            end if;
            if entered_symbols(i + 3) = "10" then
                match_count := match_count + 1;
            end if;
            if entered_symbols(i + 4) = "11" then
                match_count := match_count + 1;
            end if;

            if match_count = 5 then
                sequence_detected <= '1';
                exit;
            end if;
        end loop;
    end if;

    -- Set LED blinking patterns
    if sequence_detected = '1' then
        flash_pattern <= (others => flash_clk); -- Blink `11111111` and `00000000`
    elsif stop_flag = '1' then
        if flash_clk = '1' then
            flash_pattern <= "10101010";
        else
            flash_pattern <= "01010101";
        end if;
    else
        flash_pattern <= std_logic_vector(to_unsigned(total_count, 8)); -- Default to count
    end if;

    led_output <= flash_pattern;
  end if;
end process;

-- Assign LED output
led <= led_output;

end Behavioral;
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXX
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debouncer is
    Port (
```

```
        clk    : in std_logic;
        input  : in std_logic;
        output : out std_logic
    );
end Debouncer;

architecture Behavioral of Debouncer is
    signal delay1, delay2, delay3 : std_logic := '0';
begin
    process(clk)
    begin
        if rising_edge(clk) then
            delay1 <= input;
            delay2 <= delay1;
            delay3 <= delay2;
        end if;
    end process;

    output <= delay2 and not delay3; -- Detect stable rising edge
end Behavioral;

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXClockadjustmentXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CombinedSystem is
    Port (
        clk    : in  STD_LOGIC;              -- Clock signal
        btnc   : in  STD_LOGIC;              -- Reset button
        btnu   : in  STD_LOGIC;              -- Button Up
        btnd   : in  STD_LOGIC;              -- Button Down
        btnl   : in  STD_LOGIC;              -- Button Left
        btnr   : in  STD_LOGIC;              -- Button Right
        led    : out STD_LOGIC_VECTOR(7 downto 0) -- LED output
    );
end CombinedSystem;

architecture Behavioral of CombinedSystem is
    -- Constants
    constant MAX_ATTEMPTS      : integer := 15; -- Maximum button presses stored
    constant SEQUENCE_LENGTH   : integer := 5;
    constant CLK_DIVIDER       : integer := 3000000; -- Slow clock frequency divider
    constant FLASH_DIVIDER     : integer := 2000000; -- Flash clock frequency divider

    -- Button Press Counter
    signal total_count : INTEGER range 0 to 255 := 0; -- Total count of button presses
    signal stop_flag   : STD_LOGIC := '0'; -- Stops counting at 11
```

```
    -- Sequence Detection
    type SymbolArray is array(0 to MAX_ATTEMPTS - 1) of std_logic_vector(1 downto 0);
    signal entered_symbols : SymbolArray := (others => "00");
    signal current_index   : integer range 0 to MAX_ATTEMPTS := 0; -- Tracks current input
position
    signal sequence_detected : STD_LOGIC := '0'; -- Indicates sequence is found

    -- Clock Divider
    signal slow_clk  : STD_LOGIC := '0';
    signal flash_clk : STD_LOGIC := '0'; -- Blinking clock
    signal clk_count : integer := 0;
    signal flash_count : integer := 0;

    -- Debouncer Outputs
    signal btnu_debounced, btnd_debounced, btnl_debounced, btnr_debounced,
btnc_debounced : std_logic;

    -- LED Output
    signal led_output    : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    signal flash_pattern : STD_LOGIC_VECTOR(7 downto 0) := (others => '0'); -- Temporary
pattern for blinking
begin
    -- Clock Divider Process
    process(clk)
    begin
        if rising_edge(clk) then
            -- Generate slow clock
            if clk_count = CLK_DIVIDER - 1 then
                slow_clk <= not slow_clk;
                clk_count <= 0;
            else
                clk_count <= clk_count + 1;
            end if;

            -- Generate flash clock for blinking
            if flash_count = FLASH_DIVIDER - 1 then
                flash_clk <= not flash_clk;
                flash_count <= 0;
            else
                flash_count <= flash_count + 1;
            end if;
        end if;
    end process;

    -- Instantiate Debouncer for Each Button
    debounce_btnu: entity work.Debouncer
        Port map (
            clk    => slow_clk, -- Use slow clock
            input  => btnu,
```

```
        output => btnu_debounced
    );

    debounce_btnd: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
        input  => btnd,
        output => btnd_debounced
    );

    debounce_btnl: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
        input  => btnl,
        output => btnl_debounced
    );

    debounce_btnr: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
        input  => btnr,
        output => btnr_debounced
    );

    debounce_btnc: entity work.Debouncer
    Port map (
        clk    => slow_clk, -- Use slow clock
        input  => btnc,
        output => btnc_debounced
    );

    -- Counter and Sequence Detection Process
    process(slow_clk)
        variable match_count : integer := 0;  -- Tracks matching symbols
    begin
        if rising_edge(slow_clk) then
            if btnc_debounced = '1' then
                -- Reset system
                total_count <= 0;
                stop_flag <= '0';
                current_index <= 0;
                entered_symbols <= (others => "00");
                sequence_detected <= '0';
                led_output <= (others => '0');
            elsif sequence_detected = '0' then
                -- Count button presses
                if stop_flag = '0' then
                    if btnu_debounced = '1' or btnd_debounced = '1' or btnl_debounced = '1' or
btnr_debounced = '1' then
                        total_count <= total_count + 1;
```

```
        end if;

        -- Stop counting at 11
        if total_count = 11 then
            stop_flag <= '1';
        end if;
    end if;

    -- Record button press for sequence detection
    if btnu_debounced = '1' then
        entered_symbols(current_index) <= "00";
        current_index <= current_index + 1;
    elsif btnd_debounced = '1' then
        entered_symbols(current_index) <= "01";
        current_index <= current_index + 1;
    elsif btnl_debounced = '1' then
        entered_symbols(current_index) <= "10";
        current_index <= current_index + 1;
    elsif btnr_debounced = '1' then
        entered_symbols(current_index) <= "11";
        current_index <= current_index + 1;
    end if;

    -- Check for target sequence in any position
    for i in 0 to MAX_ATTEMPTS - SEQUENCE_LENGTH loop
        match_count := 0;
        if entered_symbols(i) = "00" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 1) = "01" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 2) = "01" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 3) = "10" then
            match_count := match_count + 1;
        end if;
        if entered_symbols(i + 4) = "11" then
            match_count := match_count + 1;
        end if;

        if match_count = 5 then
            sequence_detected <= '1';
            exit;
        end if;
    end loop;
end if;

-- Set LED blinking patterns
```

```
            if sequence_detected = '1' then
                flash_pattern <= (others => flash_clk); -- Blink `11111111` and `00000000`
            elsif stop_flag = '1' then
                if flash_clk = '1' then
                    flash_pattern <= "10101010";
                else
                    flash_pattern <= "01010101";
                end if;
            else
                flash_pattern <= std_logic_vector(to_unsigned(total_count, 8)); -- Default to count
            end if;

            led_output <= flash_pattern;
        end if;
    end process;

    -- Assign LED output
    led <= led_output;

end Behavioral;
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debouncer is
    Port (
        clk    : in std_logic;
        input  : in std_logic;
        output : out std_logic
    );
end Debouncer;

architecture Behavioral of Debouncer is
    signal delay1, delay2, delay3 : std_logic := '0';
begin
    process(clk)
    begin
        if rising_edge(clk) then
            delay1 <= input;
            delay2 <= delay1;
            delay3 <= delay2;
        end if;
    end process;

    output <= delay2 and not delay3; -- Detect stable rising edge
end Behavioral;
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
```

Appendix 2

```
##############################Inital implements code ##############################
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SequenceDetector is
    Port (
        clk     : in  STD_LOGIC;              -- Clock signal
        btnc    : in  STD_LOGIC;              -- Initialization button (middle)
        btnu    : in  STD_LOGIC;              -- Button Up
        btnd    : in  STD_LOGIC;              -- Button Down
        btnl    : in  STD_LOGIC;              -- Button Left
        btnr    : in  STD_LOGIC;              -- Button Right
        led     : out STD_LOGIC_VECTOR(7 downto 0) -- LED output
    );
end SequenceDetector;

architecture Behavioral of SequenceDetector is
    -- Constants
    constant MAX_ATTEMPTS : integer := 10; -- Max number of button presses stored
    constant SEQUENCE_LENGTH : integer := 5;
    constant LOCK_PATTERN : STD_LOGIC_VECTOR(7 downto 0) := "10101010"; --
Predefined locked pattern

    -- Target sequence definition (btnu, btnd, btnd, btnl, btnr)
    constant TARGET_SEQUENCE : std_logic_vector(1 downto 0) := "00";
    constant TARGET_SEQUENCE_ARRAY : std_logic_vector(9 downto 0) :=
"0010011011"; -- Encoded

    -- Signal declarations
    type SymbolArray is array(0 to MAX_ATTEMPTS - 1) of std_logic_vector(1 downto 0);
    signal entered_symbols : SymbolArray := (others => "00");
    signal current_index : integer range 0 to MAX_ATTEMPTS := 0; -- Tracks current input
position
    signal sequence_detected : STD_LOGIC := '0'; -- Indicates sequence is found
    signal locked : STD_LOGIC := '0'; -- Indicates system is locked
    signal flash_clk : STD_LOGIC := '0'; -- Flashing clock for LEDs
    signal flash_count : integer := 0;

    -- Button debounce signals
    signal btnu_stable, btnd_stable, btnl_stable, btnr_stable, btnc_stable : STD_LOGIC := '0';
    signal btnu_debounce, btnd_debounce, btnl_debounce, btnr_debounce, btnc_debounce :
integer range 0 to 1000000 := 0;
    constant DEBOUNCE_THRESHOLD : integer := 500000; -- Debounce threshold

    -- Clock divider
    signal slow_clk : STD_LOGIC := '0';
    signal clk_count : integer := 0;
    constant CLK_DIVIDER : integer := 10000000; -- Slower clock for processing
```

```
begin
    -- Clock divider process
    process(clk)
    begin
        if rising_edge(clk) then
            if clk_count = CLK_DIVIDER then
                slow_clk <= not slow_clk;
                clk_count <= 0;
            else
                clk_count <= clk_count + 1;
            end if;

            -- Flash clock for LEDs
            if flash_count = CLK_DIVIDER / 2 then
                flash_clk <= not flash_clk;
                flash_count <= 0;
            else
                flash_count <= flash_count + 1;
            end if;
        end if;
    end process;

    -- Debounce process for stable button signals
    process(clk)
    begin
        if rising_edge(clk) then
            -- Debounce btnu
            if btnu = '1' then
                if btnu_debounce < DEBOUNCE_THRESHOLD then
                    btnu_debounce <= btnu_debounce + 1;
                else
                    btnu_stable <= '1';
                end if;
            else
                btnu_debounce <= 0;
                btnu_stable <= '0';
            end if;

            -- Debounce btnd
            if btnd = '1' then
                if btnd_debounce < DEBOUNCE_THRESHOLD then
                    btnd_debounce <= btnd_debounce + 1;
                else
                    btnd_stable <= '1';
                end if;
            else
                btnd_debounce <= 0;
                btnd_stable <= '0';
            end if;
```

```
        -- Debounce btnl
        if btnl = '1' then
            if btnl_debounce < DEBOUNCE_THRESHOLD then
                btnl_debounce <= btnl_debounce + 1;
            else
                btnl_stable <= '1';
            end if;
        else
            btnl_debounce <= 0;
            btnl_stable <= '0';
        end if;

        -- Debounce btnr
        if btnr = '1' then
            if btnr_debounce < DEBOUNCE_THRESHOLD then
                btnr_debounce <= btnr_debounce + 1;
            else
                btnr_stable <= '1';
            end if;
        else
            btnr_debounce <= 0;
            btnr_stable <= '0';
        end if;

        -- Debounce btnc
        if btnc = '1' then
            if btnc_debounce < DEBOUNCE_THRESHOLD then
                btnc_debounce <= btnc_debounce + 1;
            else
                btnc_stable <= '1';
            end if;
        else
            btnc_debounce <= 0;
            btnc_stable <= '0';
        end if;
    end if;
end process;

-- Sequence input and detection process
process(slow_clk)
begin
    if rising_edge(slow_clk) then
        if btnc_stable = '1' then
            -- Reset system
            current_index <= 0;
            entered_symbols <= (others => "00");
            sequence_detected <= '0';
            locked <= '0';
        elsif current_index < MAX_ATTEMPTS then
            -- Record button press
```

```
            if btnu_stable = '1' then
                entered_symbols(current_index) <= "00";
                current_index <= current_index + 1;
            elsif btnd_stable = '1' then
                entered_symbols(current_index) <= "01";
                current_index <= current_index + 1;
            elsif btnl_stable = '1' then
                entered_symbols(current_index) <= "10";
                current_index <= current_index + 1;
            elsif btnr_stable = '1' then
                entered_symbols(current_index) <= "11";
                current_index <= current_index + 1;
            end if;

            -- Check for sequence
            for i in 0 to MAX_ATTEMPTS - SEQUENCE_LENGTH loop
                if entered_symbols(i) = "00" and
                   entered_symbols(i + 1) = "01" and
                   entered_symbols(i + 2) = "01" and
                   entered_symbols(i + 3) = "10" and
                   entered_symbols(i + 4) = "11" then
                    sequence_detected <= '1';
                    locked <= '0';
                    exit;
                end if;
            end loop;

            -- Lock system if full and no sequence found
            if current_index = MAX_ATTEMPTS and sequence_detected = '0' then
                locked <= '1';
            end if;
        end if;
    end if;
end process;

-- LED output control
process(flash_clk)
begin
    if sequence_detected = '1' then
        -- Flash LEDs
        led <= (others => flash_clk);
    elsif locked = '1' then
        -- Show locked pattern
        led <= LOCK_PATTERN;
    else
        -- Default: show number of inputs
        led <= std_logic_vector(to_unsigned(current_index, 8));
    end if;
end process;
```

```
end Behavioral;
```