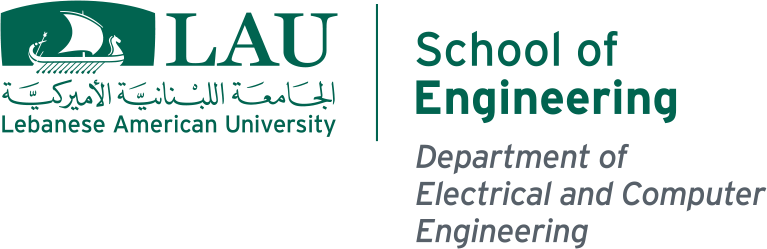
**Lebanese American University**



***COE322 – Logic Design Lab***

***Logic Design Lab Project - Logic-Controlled Board***

Layane Kanso 202304332

Elissa Saad 202301670

Hayfa Siblini 202301917

May 14, 2025

Table of Contents

[Table of Figures 3](#_Toc198091910)

[Table of Tables 4](#_Toc198091911)

[Abstract 5](#_Toc198091912)

[Introduction 6](#_Toc198091913)

[Analysis 7](#_Toc198091914)

[A. Project Overview 9](#_Toc198091915)

[*Part 1: LogicLED - Initial Sequence Mapping* 9](#_Toc198091916)

[*Part 2: Registers - Storing LED States* 11](#_Toc198091917)

[*Part 3: TestReg - Encoding the Switch States* 12](#_Toc198091918)

[*Part 4: Shift Register - Sequencing LED Activation* 13](#_Toc198091919)

[*Part 5: Sequence Detector - Identifying Active Sequence* 14](#_Toc198091920)

[*Part 6: MUXenable - Reconfiguring Register Enables Based on Sequence* 16](#_Toc198091921)

[Breadboard design and analysis 25](#_Toc198091922)

[Financial study 28](#_Toc198091923)

[Delay analysis 30](#_Toc198091924)

[Power consumption analysis 32](#_Toc198091925)

[Problems faced during the project 34](#_Toc198091926)

[Key design points that present advantages over alternative designs 41](#_Toc198091927)

[Conclusion 42](#_Toc198091928)

Table of Figures

[Figure 1 – Overall Logic Circuit 9](#_Toc198091957)

[Figure 2 - Circuit inside of LogicLED blocks 10](#_Toc198091958)

[Figure 3 - Circuit inside of Reg Blocks 11](#_Toc198091959)

[Figure 4 - Circuit inside of TestReg Block 12](#_Toc198091960)

[Figure 5 - Circuit inside of Counter block 13](#_Toc198091961)

[Figure 6 - Circuit inside of SeqDet Block 14](#_Toc198091962)

[Figure 7 - Circuit inside of NewEnable Block 16](#_Toc198091963)

[Figure 8 - Circuit inside of MUXEnable Block 17](#_Toc198091964)

[Figure 9 - Pulse\_gen code 18](#_Toc198091965)

[Figure 10 - Pulse\_gen block 19](#_Toc198091966)

[Figure 11 - Initial condition locked state 21](#_Toc198091967)

[Figure 12 - ModelSim simulation 23](#_Toc198091968)

[Figure 13 - 555 timer datasheet 26](#_Toc198091969)

[Figure 14 - Breadboard circuit 27](#_Toc198091970)

[Figure 15 – Delay 30](#_Toc198091971)

[Figure 16 - Timestamp of input transition in the simulation 31](#_Toc198091972)

[Figure 17 - Timestamp of output transition in the simulation 31](#_Toc198091973)

[Figure 18 - Sequence Detector Circuit Using D Flip Flops (Without SR Latches) 34](#_Toc198091974)

[Figure 19 - Simulation of Circuit Without SR Latches 35](#_Toc198091975)

[Figure 20 - Modified Sequence Detector Circuit Using SR Latches 35](#_Toc198091976)

[Figure 21 - State Diagram for FSM-Based Sequence Detector 37](#_Toc198091977)

[Figure 22 - Example of a Long Output Equation 39](#_Toc198091978)

Table of Tables

[Table 1 - Table to encode our sequence bits 14](#_Toc198092117)

[Table 2 - Bill of Materials with component quantities and costs 28](#_Toc198092118)

[Table 3- State Table for FSM-Based Sequence Detector 37](#_Toc198092119)

# **Abstract**

This project focuses on the design and implementation of a Logic Controlled Board that responds dynamically to user interaction. The system consists of four toggle switches and four LEDs of corresponding colors. Unlike static logic designs, this system adapts its behavior based on the last switch turned off, activating a unique LED sequence. The design uses finite state machines, combinational logic, and timing circuits, along with special interactive features such as temporary switch disabling and sequence locking. The project was simulated using Quartus software and implemented physically on a breadboard. The final outcome demonstrates how theoretical digital logic concepts can be effectively applied in real-world applications.

Introduction

The Logic Controlled Board project serves as a practical application of the digital logic concepts we studied throughout the semester. It allowed us to design and implement a system that integrates combinational and sequential logic, finite state machines, and timing circuits. The board consists of four switches and four LEDs, but unlike a simple static mapping, it changes behavior based on the last switch turned off. We incorporated interactive features such as sequence-based LED control, a timed reset, switch cap functionality, and a practice mode with a seven segment display. This project enabled us to bridge theory and practice by building a complete, responsive system from design to physical implementation.

# Analysis

The Logic Controlled Board is a digital system we designed to control the behavior of four colored LEDs based on how four toggle switches are used. While the system may appear simple at first glance, it functions in a dynamic and logical manner. Instead of having each switch permanently mapped to its corresponding LED, the system detects the last switch that was turned off and reassigns the LED activation sequence accordingly. This design creates the illusion that the board responds intelligently or remembers past actions.

The key idea behind the project is sequence-based control. Each time the system resets, it activates a new LED sequence based on the last switch that was turned off before all switches were turned off. This reset is triggered after a four second delay, allowing the board to determine which switch was deactivated last. Once the reset condition is met, the system enters a new sequence mode, and the LEDs respond based on that configuration until the next reset occurs.

To better understand the logic behind the system, it is important to examine the inputs, outputs, and sequence behavior in detail.

**Inputs:**

The system relies on the following inputs to function:

* **SW1, SW2, SW3, SW4**: These are the four toggle switches on the board. They are the only points of direct user interaction. The system continuously monitors their ON and OFF states to determine which LED should be active and when a reset should occur.
* **Reset condition**: This is a logical condition, not a physical input. When all four switches are OFF and remain so for four seconds, the system resets and determines which sequence to activate based on the last switch that was turned OFF.

**Outputs:**

The outputs of the system provide immediate visual feedback and assist in debugging during development:

* **L1, L2, L3, L4**: These four LEDs are each assigned a color (red, green, blue, and yellow). Their behavior is determined by the active sequence, meaning their response to the switches changes depending on which sequence is selected.
* **Seven segment display**: This optional output was used during development to display the current sequence number (1 to 4). It helped us confirm that the correct sequence was selected after each reset. While useful in practice mode, it can be removed in performance mode for a cleaner appearance.

**LED Activation Sequences:**

The core functionality of the Logic Controlled Board lies in the way it dynamically changes the LED control pattern based on the last switch turned off. Each switch is assigned to a specific LED activation sequence as follows:

* **If SW1 was turned off last**:  
  Sequence 1 is activated  
  LED order: 1 → 2 → 3 → 4  
  SW1 controls L1, SW2 controls L2, SW3 controls L3, SW4 controls L4
* **If SW2 was turned off last**:  
  Sequence 2 is activated  
  LED order: 2 → 3 → 4 → 1  
  SW1 controls L2, SW2 controls L3, SW3 controls L4, SW4 controls L1
* **If SW3 was turned off last**:  
  Sequence 3 is activated  
  LED order: 3 → 4 → 1 → 2  
  SW1 controls L3, SW2 controls L4, SW3 controls L1, SW4 controls L2
* **If SW4 was turned off last**:  
  Sequence 4 is activated  
  LED order: 4 → 3 → 2 → 1  
  SW1 controls L4, SW2 controls L3, SW3 controls L2, SW4 controls L1

These sequences ensure that the board does not behave in a fixed or predictable way. Instead, it responds to the user's most recent actions and reorganizes itself accordingly. This gives the impression of a smart system, although it is fully based on logic and memory tracking.

The design was not built using formal truth tables or state diagrams. Instead, we approached the system by thinking through the logic step by step. We planned the sequences manually, tested the responses, and built the logic directly into the circuit. This method allowed us to stay focused on functionality and keep the design intuitive and reliable.

Overall, the Logic Controlled Board behaves as a responsive digital system that adapts to how the user interacts with it. The LED control logic is clear and consistent, and the dynamic behavior is managed entirely through a combination of switch states, memory, and timed resets. This analysis outlines the thought process and structure that made the system function as intended.

## A. Project Overview

This project implements a multi-functional digital circuit system that dynamically maps switch inputs to LEDs, supports multiple programmable sequences, and maintains internal state through register-based memory. The system is modular, scalable, and designed to operate through synchronized clock pulses, with dedicated logic for initialization and state locking.

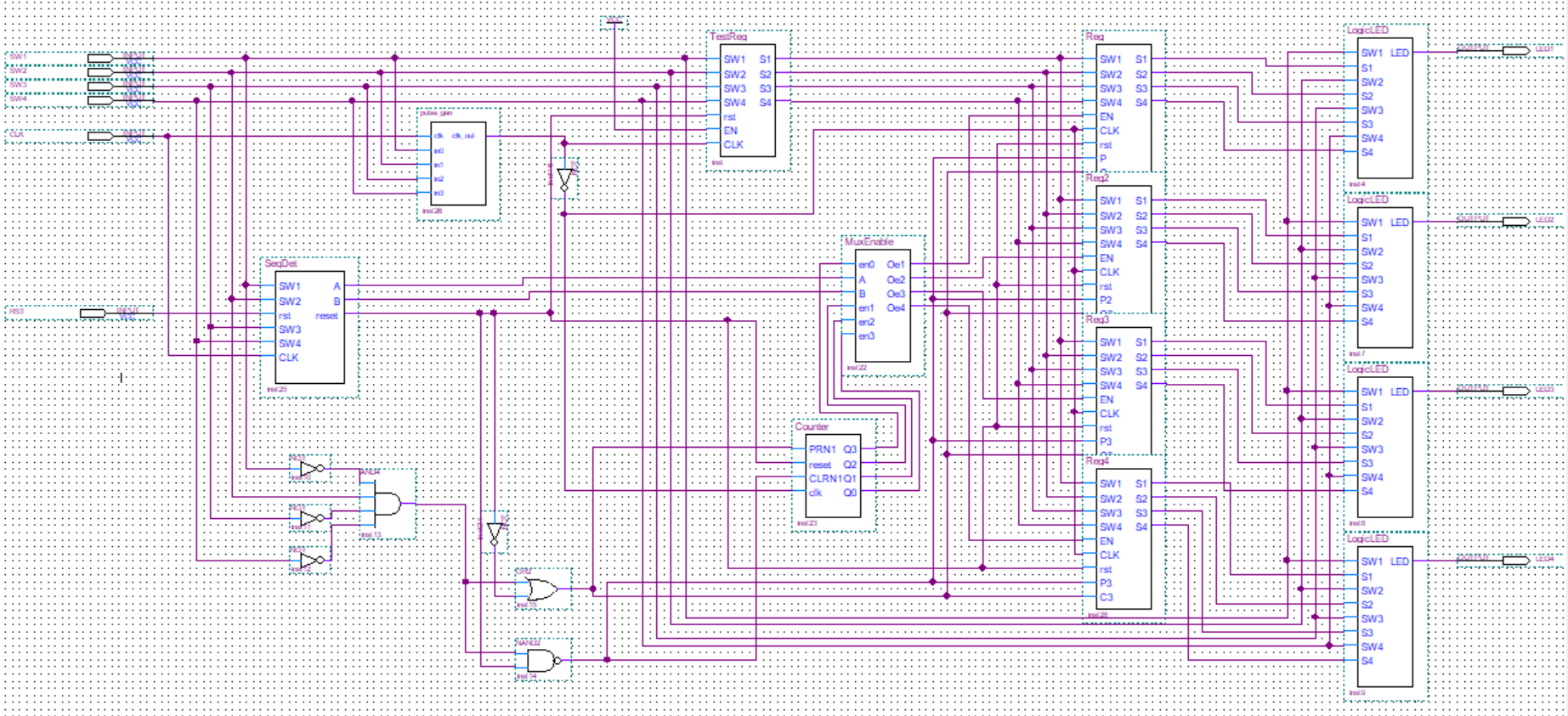


Figure 1 – Overall Logic Circuit

Part 1: LogicLED - Initial Sequence Mapping

We began by creating Sequence 1, which turns on LED1, then LED2, LED3, and finally LED4, regardless of which switches (SW1 to SW4) are used. A key requirement was that each switch, when used to activate an LED, would also be responsible for turning it off. This mapping was critical to ensure intuitive interaction between the hardware components.

To implement this, we defined a 4-bit switch encoding: **S1S2S3S4**. We adopted a convention where a switch in the ON state is represented by '0' and in the OFF state by '1'. For example:

* SW1 ON: 0111
* SW2 ON: 1011
* SW3 ON: 1101
* SW4 ON: 1110

This encoding ensures the detection of an activated switch using logic that aligns with the behavior of the logic gates.

In the LogicLED block, the output for LED1 is computed as:

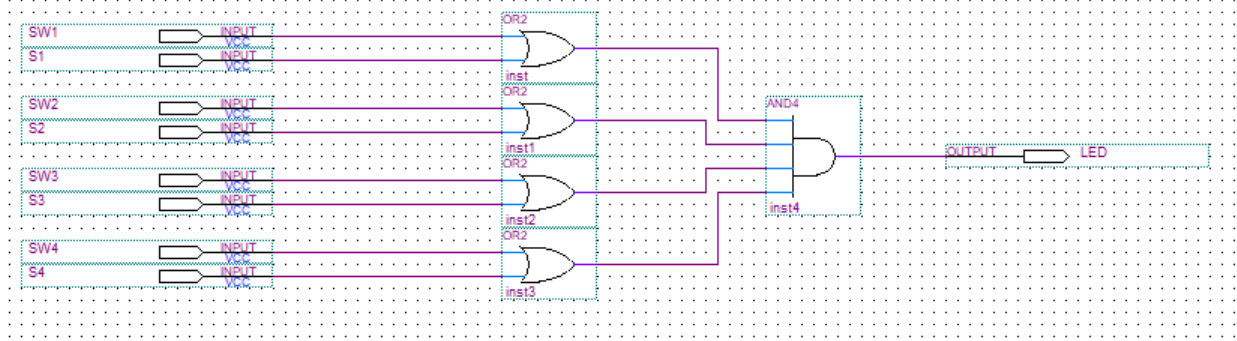
****

Figure 2 - Circuit inside of LogicLED blocks

**Example:** If we activate SW3 (S1S2S3S4 = 1101):

* LED1 = (1 + SW1)(1 + SW2)(0 + SW3)(1 + SW4)
* Simplifying: LED1 = 1 \* 1 \* SW3 \* 1 = SW3

Thus, LED1 is directly controlled by SW3. Turning SW3 on turns the LED on, and turning it off switches it off. This achieved the required switch-to-LED mapping. This also goes for the other 3 leds.

Part 2: Registers - Storing LED States

To preserve the encoded switch bits, we used two 7474 dual D-type flip-flops (totaling 4 bits). These flip-flops store the S1S2S3S4 state and feed their outputs into the LogicLED block.

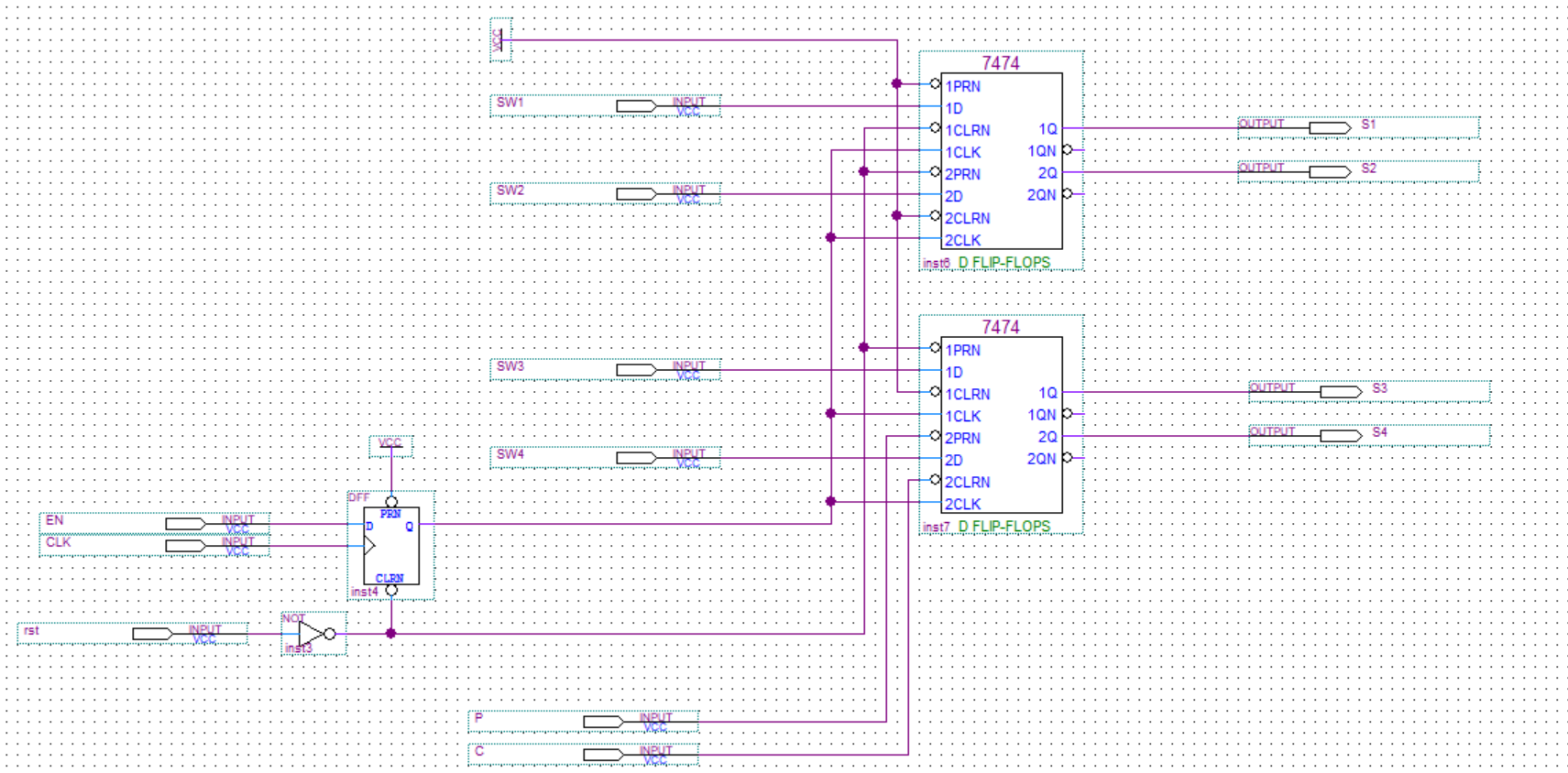


Figure 3 - Circuit inside of Reg Blocks

Inputs to the registers are the encoded bits, which we explain in Part 3. The registers are clocked by the pulse generator, but due to timing issues observed during simulation, we introduced an additional D flip-flop between the pulse generator and the 7474s to introduce a controlled delay.

Presets and clears of the 7474s are reserved for handling the locked state, which we discuss in a later section.

Part 3: TestReg - Encoding the Switch States

To generate the S1S2S3S4 encoded bits, we created the TestReg block. This module uses two 7474 flip-flops in cascade, where the output of the first is NXORed with the output of the second. This allows us to detect changes in switch states.

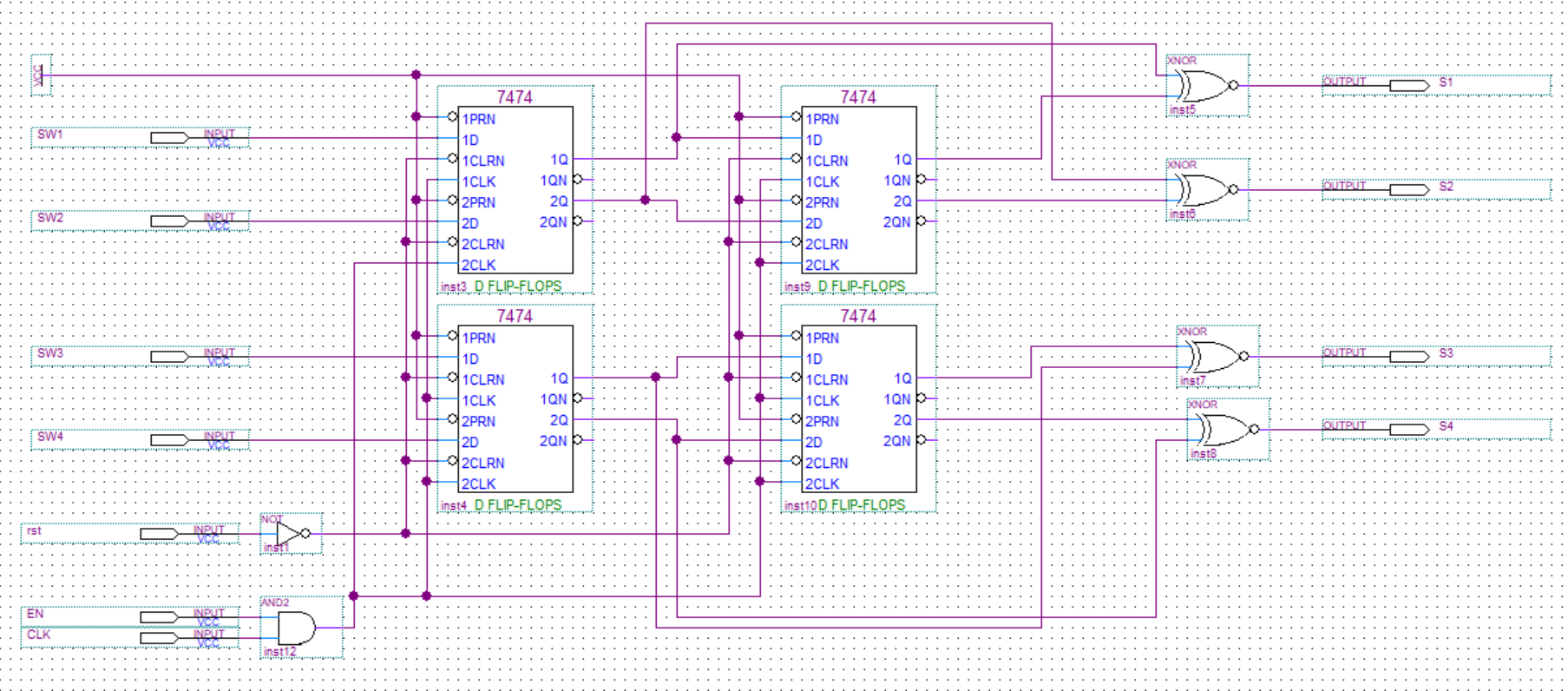


Figure 4 - Circuit inside of TestReg Block

**Example:**

* Initial switch state: 0000 → No switches ON
* After turning on SW1: 1000 → Desired output: 0111
* After turning on SW3 next: 1010 → Desired output: 1101

This behavior mirrors the NXOR gate truth table, where the output is ‘1’ when both inputs are the same and ‘0’ otherwise. We NXOR the previous and current switch states to detect which switch was activated.

All presets are tied to logic '1', and clears are tied to a global reset signal (**\_Reset**), which plays a critical role and is discussed later.

Part 4: Shift Register - Sequencing LED Activation

With mapping established, we next required a mechanism to sequence the register activations—specifically to follow the order 1 → 2 → 3 → 4.

We constructed a 4-bit shift register using two 7474s with an initial state of 1000. The D input of the first flip-flop is tied to ground, so each clock pulse shifts the ‘1’ rightward while filling with ‘0’s from the left:

**Pattern:** 1000 → 0100 → 0010 → 0001 → 0000

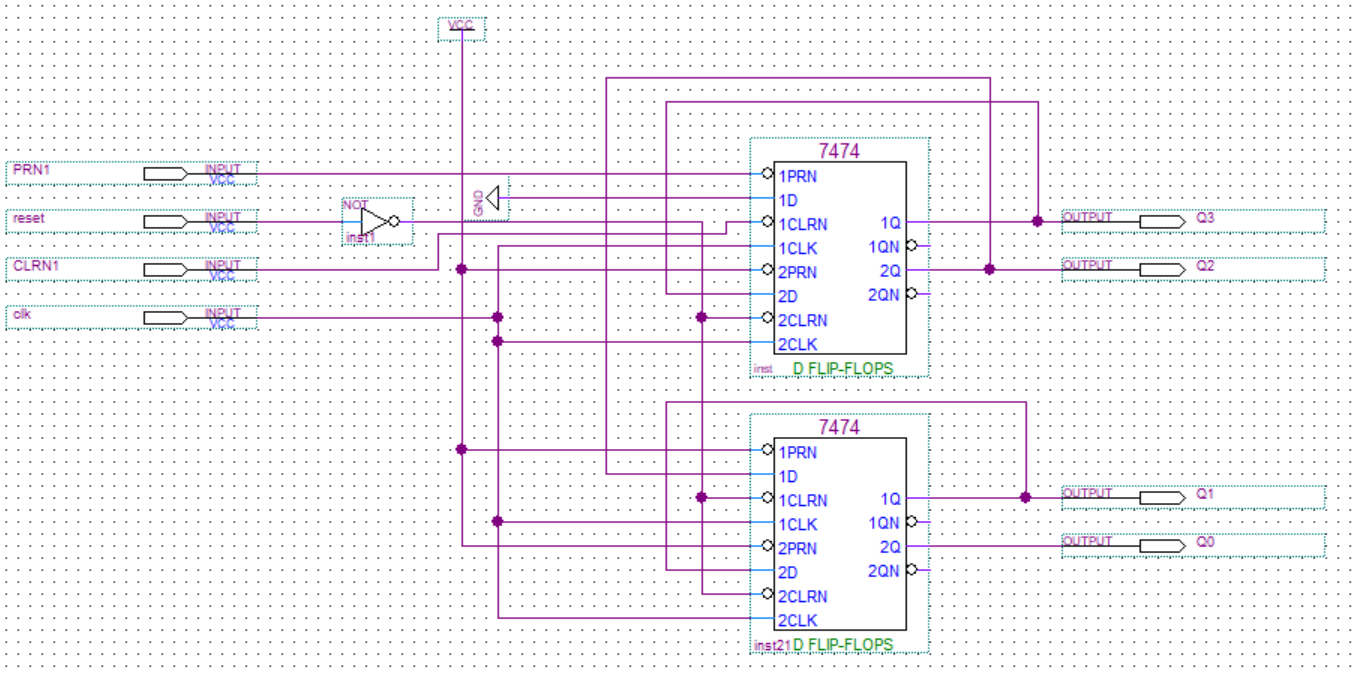


Figure 5 - Circuit inside of Counter block

This behavior enables the sequential activation of the LEDs in Sequence 1 and ensures a controlled progression.

Part 5: Sequence Detector - Identifying Active Sequence

To support multiple sequences, we needed a way to determine which sequence was active. This was based on identifying which switch was turned OFF last. We reused the TestReg logic but inverted the input switches using a 7404 inverter chip. This allowed us to detect the last switch-off event.

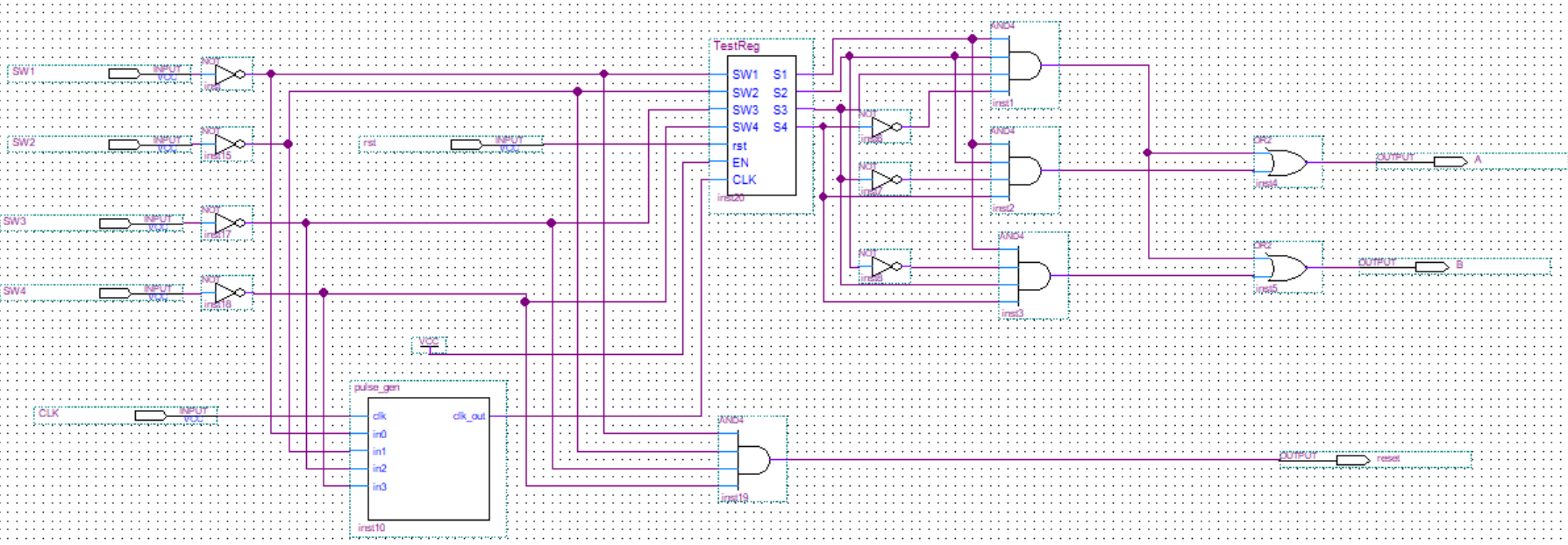
****

Figure 6 - Circuit inside of SeqDet Block

**Example:**

* If SW1 is last off: S1S2S3S4 = 0111

We defined the following table to decode the sequence:

Table 1 - Table to encode our sequence bits

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S1** | **S2** | **S3** | **S4** | **A** | **B** |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 |

Using sum of minterms, we derived:

The outputs A and B represent a 2-bit sequence code and are connected to a BCD-to-7 segment display (common anode) to indicate the current sequence.

Additionally, the sequence detector block implements the global reset logic for the entire circuit. Rather than relying on an external push-button reset, the system is designed to reset itself when all input switches are turned off, i.e., when the input pattern is S1'S2'S3'S4'. This condition triggers a reset signal that propagates to all major modules—counters, registers, detectors—ensuring a clean return to the initial state. This design choice enhances user experience and simplifies testing by embedding reset behavior directly into the system logic.

Part 6: MUXenable - Reconfiguring Register Enables Based on Sequence

To allow different sequences to be executed, we developed the MUXenable block. It dynamically rearranges the register enables based on the current sequence identified by AB.

We defined enables for four cycles. For each cycle, enable logic expressions are as follows:

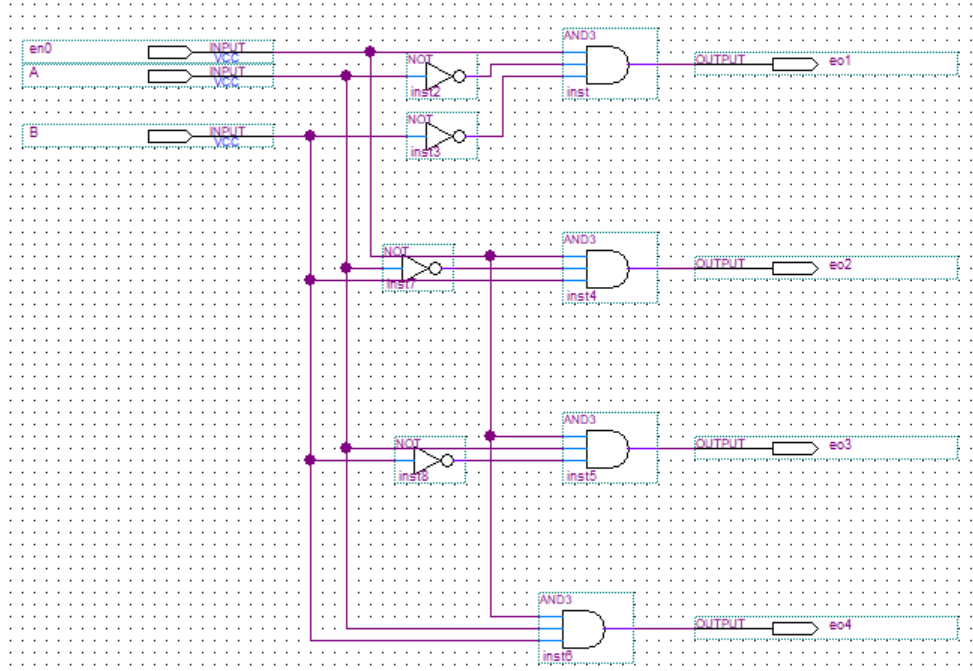


Figure 7 - Circuit inside of NewEnable Block

**Cycle 1:**

* e01 = en0·A'·B' → e1
* e02 = en0·A'·B → e2
* e03 = en0·A·B' → e3
* e04 = en0·A·B → e4

**Cycle 2:**

* e01 = en1·A'·B' → e2
* e02 = en1·A'·B → e3
* e03 = en1·A·B' → e4
* e04 = en1·A·B → e1

**Cycle 3:**

* e01 = en2·A'·B' → e3
* e02 = en2·A'·B → e4
* e03 = en2·A·B' → e1
* e04 = en2·A·B → e2

**Cycle 4:**

* e01 = en3·A'·B' → e4
* e02 = en3·A'·B → e1
* e03 = en3·A·B' → e2
* e04 = en3·A·B → e3

Each group of four outputs from the MUXenable block is combined using 4-input OR gates to produce the final register enable signals: e1, e2, e3, e4.

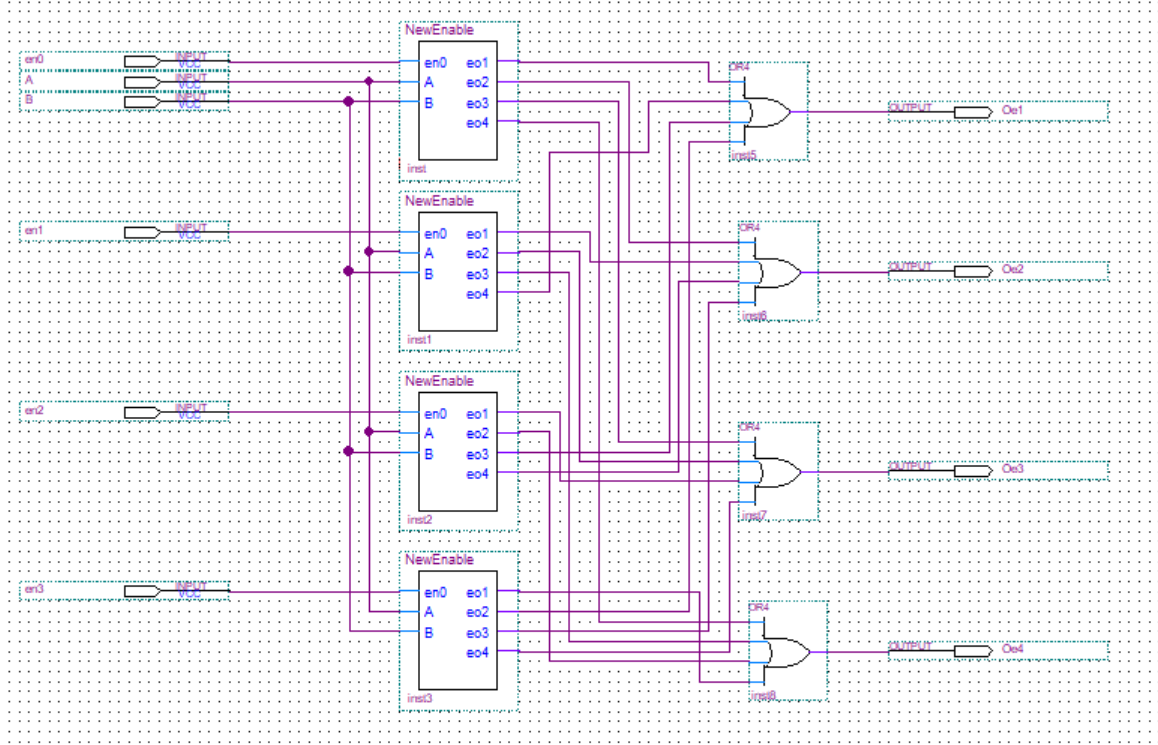


Figure 8 - Circuit inside of MUXEnable Block

This generalized enable logic allows us to scale the system from supporting just Sequence 1 to all four sequences, maintaining proper activation order based on the last switch-off.

Part 7: clock

In the design and implementation of our digital system, we replaced the conventional 555 timer typically used for pulse generation with a custom-designed VHDL-based pulse generator. This decision was intentional and strategic. While the 555 timer is effective in generating pulses in analog or hybrid designs, it lacks the synchronization and reliability required in fully digital FPGA-based systems. Therefore, to ensure more accurate and consistent timing behavior, we implemented a digital pulse generator (pulse\_gen) using VHDL.

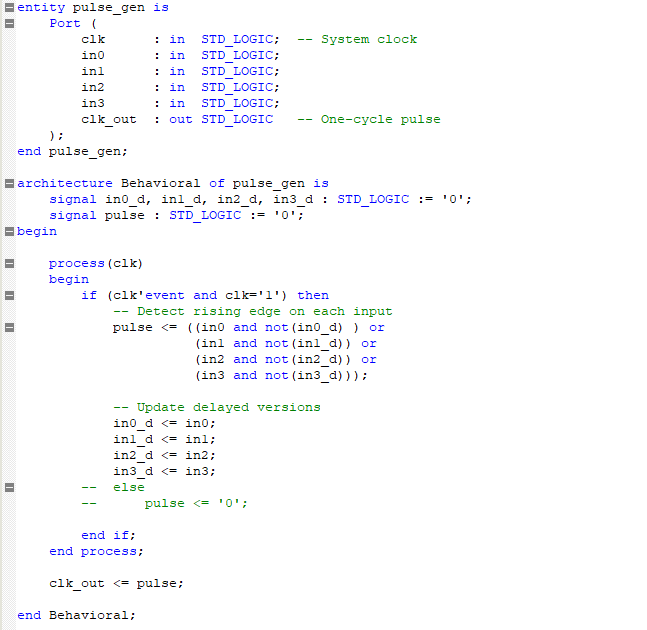


Figure 9 - Pulse\_gen code

The pulse generator is driven by the system clock (clk) and is capable of detecting rising edges on four separate inputs (in0 to in3). Whenever any of these inputs transition from low ('0') to high ('1'), the module outputs a single clock-cycle-wide pulse on the output signal (clk\_out). Internally, this is accomplished by comparing the current state of each input with its delayed version stored from the previous clock cycle, enabling precise edge detection. This ensures that only a single, clean pulse is produced per transition—eliminating the risk of glitches or signal bouncing that could occur with mechanical switches or less robust analog circuits.

This generated pulse is critical to our system's operation, as it serves as a reliable trigger for activating state transitions in the finite state machine (FSM) and for updating LED behaviors. By synchronizing the pulse to the system clock, we were able to ensure deterministic behavior across all modules. Moreover, the digital implementation allows for easier simulation, debugging, and integration within Quartus—benefiting both development speed and overall system stability.

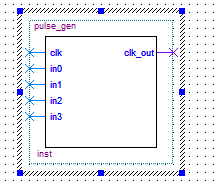


Figure 10 - Pulse\_gen block

The pulse\_gen module was thoroughly tested through waveform simulation confirming its accuracy and effectiveness in replacing traditional analog methods. Its modularity and clean design also make it reusable for future projects requiring similar input-triggered pulses.

Part 8: detect and preset

In our project, detect and preset mechanisms were essential for initializing the system state based on specific switch configurations. This functionality ensures that the system responds intelligently when powered on or reset, especially when entering different operational modes like locked or dynamic sequence modes.

Upon startup, the system reads the status of the input switches (SW1–SW4). A detection routine checks which switch was most recently turned off before a reset or power cycle. This is handled inside the Sequence Detector FSM state, which plays a crucial role in assigning one of the four predefined operation modes (Sequence 1–4). Each mode corresponds to a unique LED activation pattern based on the detected switch.

Additionally, the preset mechanism handles cases like the lock condition, where the system must initialize all outputs in a known state. If Switch 2 (SW2) is ON during boot, the system enters the Locked state, in which each switch maps directly to its corresponding LED. This bypasses the usual sequence logic and serves as a secure, predictable fallback mode.

To implement this behavior, flip-flops and state registers are used to store the detected switch state and initialize the FSM accordingly. This ensures consistent and deterministic behavior every time the system is powered on or reset, aligning with the expected mode and eliminating uncertainty due to switch bouncing or timing mismatches.

In summary, the detect and preset logic provides a robust foundation for determining the system’s starting state and configuring its initial behavior, whether operating in a dynamic or static (locked) mode. This ensures reliability and user-friendly interaction across different use scenarios.

**Activation Mechanism**

The system enters Locked State when the following initialization sequence occurs:

1. Power is applied to the system
2. **Switch 2 (SW2) is in the ON position** during power-up
3. The system detects this condition during the Boot State

**Behavior in Locked State**

* Each switch directly controls its corresponding LED:
  + SW1 → LED1 (Red)
  + SW2 → LED2 (Green)
  + SW3 → LED3 (Blue)
  + SW4 → LED4 (Yellow)
* All dynamic sequencing logic is disabled

The control logic for Locked State activation can be represented by the following truth table:

|  |  |  |  |
| --- | --- | --- | --- |
| **R** | **D** | **P** | **C** |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |

Figure 11 - Initial condition locked state

1. **Locked State ONLY occurs when R=1 AND D=1** (during reset/power-up with SW2 ON)
   * P=1 (Locked State)
   * C=0 (disables sequence logic)
2. **Normal Operation** occurs in two cases:
   * R=0, D=0: Default sequence mode
   * R=0, D=1: Still normal mode (SW2 being ON doesn't trigger lock without reset)
3. **Reset Conditions**:
   * R=1, D=0: Full system reset (P=0)
   * R=1, D=1: Reset + enter locked mode

**Your Specific Cases Revisited:**

1. "R=0, D=0: System did not reset"
   * Correct: Normal sequence mode (P=1, C=1)
2. "R=0, D=1: SW2 is ON but not locked"
   * Correct: Locked state requires reset (R=1) - this is normal operation
3. "R=1, D=0: System reset"
   * Correct: P=0 (boot state), C=0 (temporarily disabled)
4. "R=1, D=1: Locked state"
   * Correct: This is the ONLY way to enter locked mode

This ensures that Locked State is only entered when both the reset condition (R) is active (during power-up) and SW2 (D) is high. In all other cases, the system either boots normally or maintains its current operating mode. The output mode signal (C) serves as an enable/disable for the sequencing logic, with C=0 disabling all sequence generation circuits when in Locked State. This elegant solution provides robust mode control with minimal additional hardware, using just a few logic gates to implement the state transition conditions while maintaining clear separation between normal and locked operation modes.

Part 9: Simulation and Results

To validate the functionality of our complete design, we conducted a detailed simulation using ModelSim. The simulation was run step by step to observe how the system behaves under various input sequences. Below is a breakdown of the observed behavior:

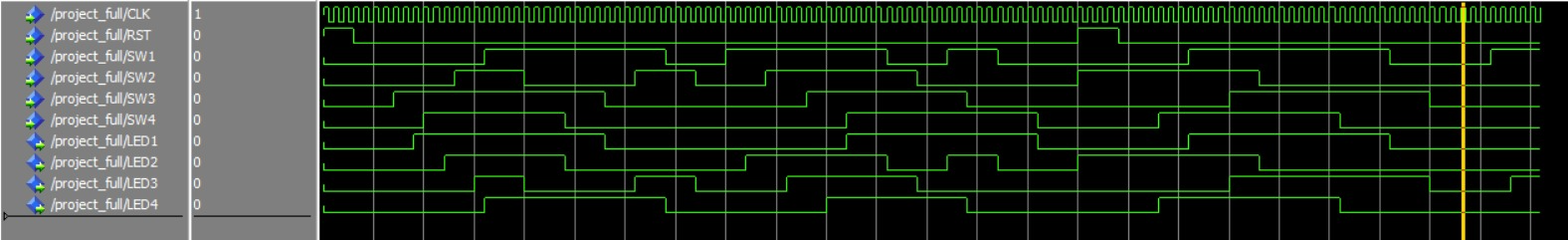


Figure 12 - ModelSim simulation

1. **Initial Reset and Default State Behavior**

At the beginning of the simulation, the reset signal was asserted while all switches were turned off. As expected, all LEDs remained off, confirming that the global reset logic and initial state were functioning correctly.

Once the reset signal was deasserted, the system transitioned into the default case, where each switch is mapped to a specific LED as follows:

* **Turning on Switch 3** → LED 1 turned ON
* **Turning on Switch 4** → LED 2 turned ON
* **Turning on Switch 2** → LED 3 turned ON
* **Turning on Switch 1** → LED 4 turned ON

Subsequently, we tested the deactivation logic by turning off the switches one by one:

* **Switch 2 turned OFF** → LED 3 turned OFF
* **Switch 4 turned OFF** → LED 2 turned OFF
* **Switch 3 turned OFF** → LED 1 turned OFF
* **Switch 2 turned ON again** → LED 3 turned ON
* **Switch 1 turned OFF** → LED 4 turned OFF
* **Switch 2 turned OFF** → LED 3 turned OFF

These results clearly verify that each switch correctly controls its corresponding LED, and the system maintains consistent behavior across toggles.

1. **Sequence Detection Behavior**

At this point, since the **last switch turned off was Switch 2**, the sequence detector recognized this and transitioned the system into **Sequence 2**, as intended. Without pressing the reset, we continued:

* **Turning on Switch 1** → LED 2 turned ON
* **Turning on Switch 2** → LED 3 turned ON
* **Turning on Switch 3** → LED 4 turned ON
* **Turning on Switch 4** → LED 1 turned ON

This confirmed the functionality of the sequence detector in correctly identifying the last switch turned off and shifting the output LED mapping accordingly.

1. **Locked State Verification**

To test the behavior after a reset, we jumped ahead in the simulation and asserted the reset signal again, this time while Switch 2 was ON. After deasserting the reset:

* **Switch 4 turned ON** → LED 4 turned ON
* **Switch 2 turned OFF** → LED 2 turned OFF
* **Switch 1 turned ON** → LED 1 turned ON
* **Switch 3 turned ON** → LED 3 turned ON

These results demonstrate that the system successfully returned to a locked state, in which each switch is mapped directly to its corresponding LED (i.e., Switch i → LED i). The consistent LED control confirms the reliability of our global reset logic and final state behavior.

## Breadboard design and analysis

To physically implement the project, we replicated the exact connections used in the Quartus simulation on the breadboard. Every wire and component placement followed the same logic design layout to ensure consistency between the simulated and real-world results. The only key modification involved the substitution of the simulated clock pulse: instead of using a virtual clock, we generated the pulse using a 555 timer circuit configured in monostable mode. This allowed us to produce a continuous square wave signal that acted as the system clock, driving the circuit through its states just as it did in the simulation. The setup ensured that the timing and logic of the system could be observed and tested in real time using physical hardware.

In our experiment, we implemented the 555 timer in monostable mode, guided by the standard configuration provided in the NE555 datasheet. In this mode, the timer generates a single output pulse of a defined duration in response to a negative trigger pulse.

**Pin Configuration and Circuit Connections:**

* **Pin 1 (GND)**: Connected to ground.
* **Pin 8 (VCC)**: Connected to a stable 5V power supply.
* **Pin 4 (RESET)**: Connected directly to VCC (5V) to disable the reset function, ensuring uninterrupted operation.
* **Pin 2 (TRIGGER)**: This pin was connected to a push-button switch. A 10 kΩ pull-up **resistor** was added between this pin and VCC to keep the trigger line HIGH by default. When the button is pressed, the trigger pin is pulled LOW, initiating the timing cycle. According to the datasheet, the trigger is activated when the voltage on this pin drops below 1/3 VCC.
* **Pin 6 (THRESHOLD)** and **Pin 7 (DISCHARGE)**: These two pins were connected together. A 36 kΩ resistor was connected between these joined pins and VCC. A 100 µF electrolytic capacitor was connected from pin 6 to ground. This resistor-capacitor (RC) combination determines the duration of the output pulse.
* **Pin 3 (OUTPUT)**: Connected to an LED (with a suitable series resistor) to visually indicate the output pulse duration. The LED turns ON when the output goes HIGH after a trigger event.
* **Pin 5 (CONTROL VOLTAGE)**: This pin was left unconnected for default threshold/reference operation, but a 10 nF (0.01 µF) ceramic capacitor was added between this pin and ground to filter out high-frequency noise, as recommended in the datasheet.

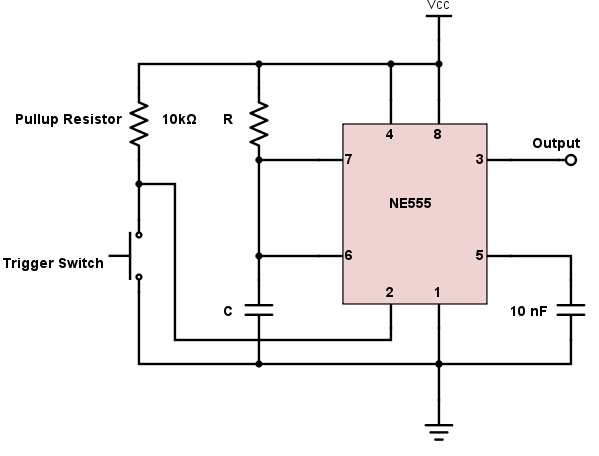


Figure 13 - 555 timer datasheet

**Component Selection and Timing Calculation:**

We chose a 36 kΩ resistor and a 100 µF capacitor to control the pulse width. The output pulse duration T in monostable mode is determined by the standard formula:

Thus, the output remains HIGH for approximately **3.96 seconds** after the trigger is activated. This duration was selected to be long enough for clear observation and measurement using an LED and oscilloscope.

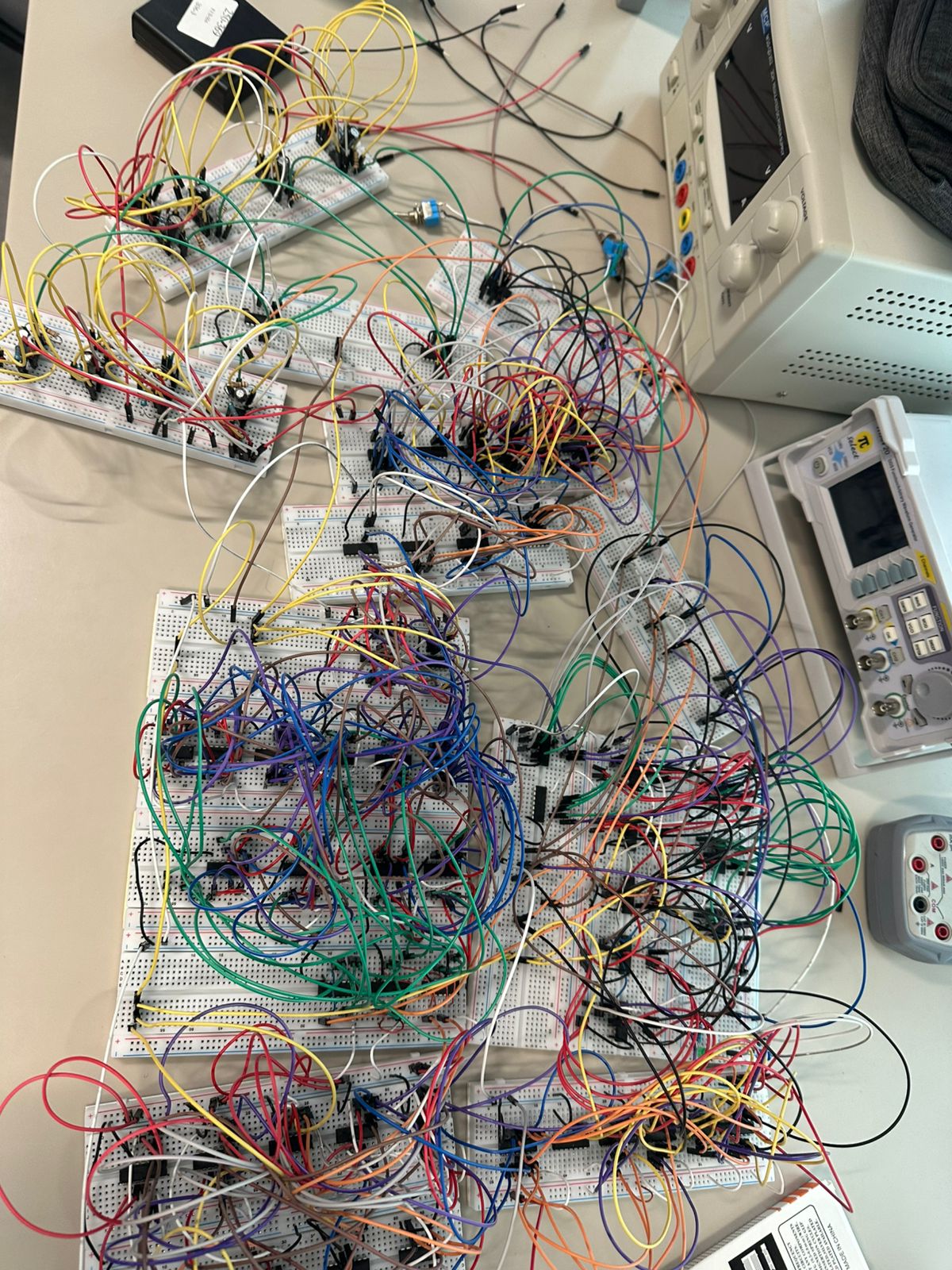


Figure 14 - Breadboard circuit

Despite our efforts to carefully follow the design, the complexity of the circuit introduced a high potential for connection errors. With numerous wires, ICs, and components placed closely together, even a single misplaced or loose wire could disrupt the entire functionality. Unfortunately, due to this intricate setup, the circuit did not function as expected during testing on the breadboard. o(╥﹏╥)o

## Financial study

The following table summarizes the electronic components utilized in the experiment, including their quantities, specifications, individual costs, and total costs. This breakdown provides a clear overview of the budget required for the breadboard implementation of the logic design project.

Table 2 - Bill of Materials with component quantities and costs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Component** | **Quantity** | **Specification** | **Unit Price (USD)** | **Total Price (USD)** |
| IC | 19 | SN7474 | 1.5 | 28.5 |
| IC | 5 | SN7404 | 0.5 | 2.50 |
| IC | 10 | SN7432 | 0.5 | 5.00 |
| IC | 4 | SN266 | 1.5 | 6.00 |
| IC | 8 | LM555 | 0.2 | 1.60 |
| IC | 8 | SN7421 | 0.5 | 4.00 |
| Resistor | 16 | 4.7KΩ | 0.02 | 0.32 |
| Resistor | 16 | 1kΩ | 0.02 | 0.32 |
| Resistor | 16 | 68Ω | 0.02 | 0.32 |
| Resistor | 4 | 220Ω | 0.02 | 0.08 |
| Breadboard | 18 | 830 Tie Point, MB-102 | 1.00 | 18.00 |
| Jumper Wires | 15 | 20cm | 0.9 | 13.5 |
| Battery | 4 | 1500mA | 1.5 | 6.00 |
| Resistor | 8 | 33kΩ | 0.02 | 0.16 |
| Capacitor | 8 | 100µF | 0.10 | 0.80 |
| Capacitor | 8 | 10µF | 0.10 | 0.80 |
| Resistor | 10 | 10kΩ | 0.02 | 0.20 |
| Switches | 4 | 85 HS-104 | 0.35 | 0.70 |
| LEDs | 4 | 5mm | 0.20 | 0.8 |

**Total: $89.60**

As shown in the table above, the total cost of the components used in the breadboard implementation amounts to $89.60. This total includes all essential parts such as ICs, passive components, power supply, and wiring. While this amount may seem relatively high for a student project, it reflects the complexity of the logic design and the number of components required to replicate the full simulation on hardware. Most of the components are reusable in future projects, which helps justify the one-time investment. Overall, the financial aspect of the project remained manageable within typical lab budgets.

## Delay analysis

To evaluate the timing behavior of our circuit, we initially attempted to analyze delays using ModelSim. However, we quickly realized that the simulation results appeared overly ideal. The outputs changed immediately in response to the inputs, with no observable propagation delay or transition lag. This is because ModelSim, in its default functional simulation mode, does not account for real hardware delays. It performs what is known as a zero delay simulation, where all logic gates and sequential elements like flip flops are assumed to operate without any time delay. The simulator simply checks for logical correctness without modeling physical behavior such as gate propagation times, wire delays, or setup and hold requirements.

In contrast, real world circuits, especially ones as complex as ours, inevitably experience noticeable delays due to the physical characteristics of the components and wiring. Our design includes many D flip flops and several layers of combinational logic gates, which makes it prone to cumulative propagation delays. As signals pass through each stage, from one flip flop through logic gates and into the next, even small individual delays can build up significantly. Additionally, the larger the number of sequential updates, such as state transitions or output activations, the more delay we expect, particularly in edge triggered circuits where timing coordination is important.

Given the structural complexity of our system, it is natural and expected that real delay would be present when implemented on actual hardware or during a post fitting timing simulation in Quartus. Therefore, while ModelSim helped verify the logical functionality of the design, it could not provide an accurate representation of the circuit’s true timing performance.

To demonstrate the expected delay in our design, we will show an example using Quartus timing simulation. This example will allow us to measure the delay between an input change and the corresponding output response, providing a more realistic view of the circuit's behavior.

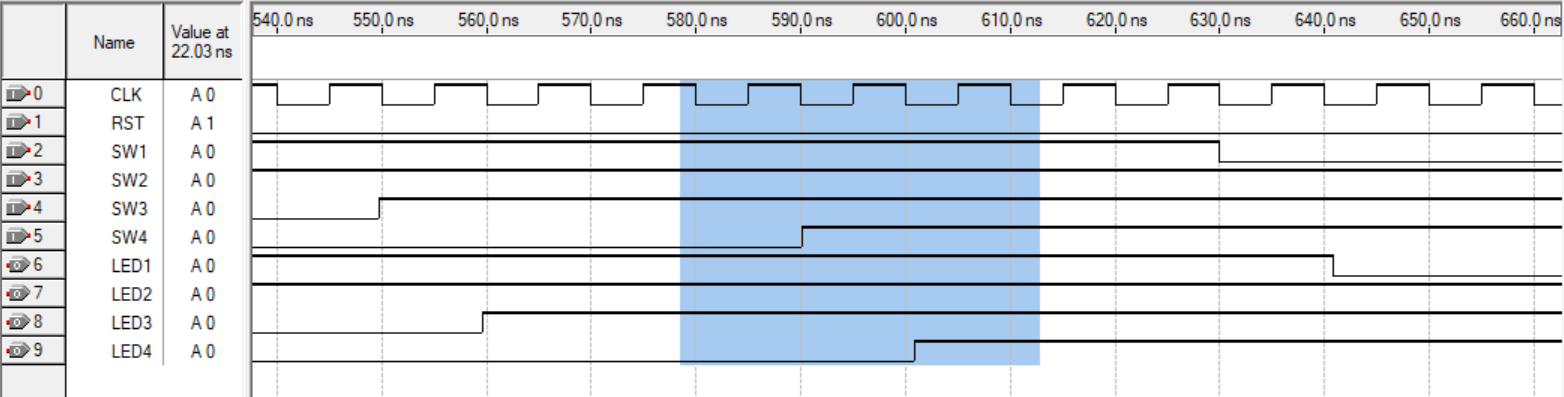


Figure 15 – Delay

Here, we can clearly see that the output does not change instantaneously when the input does. This delay is introduced due to the inherent propagation delay of the flip-flops, causing a slight shift in the timing of output transitions.

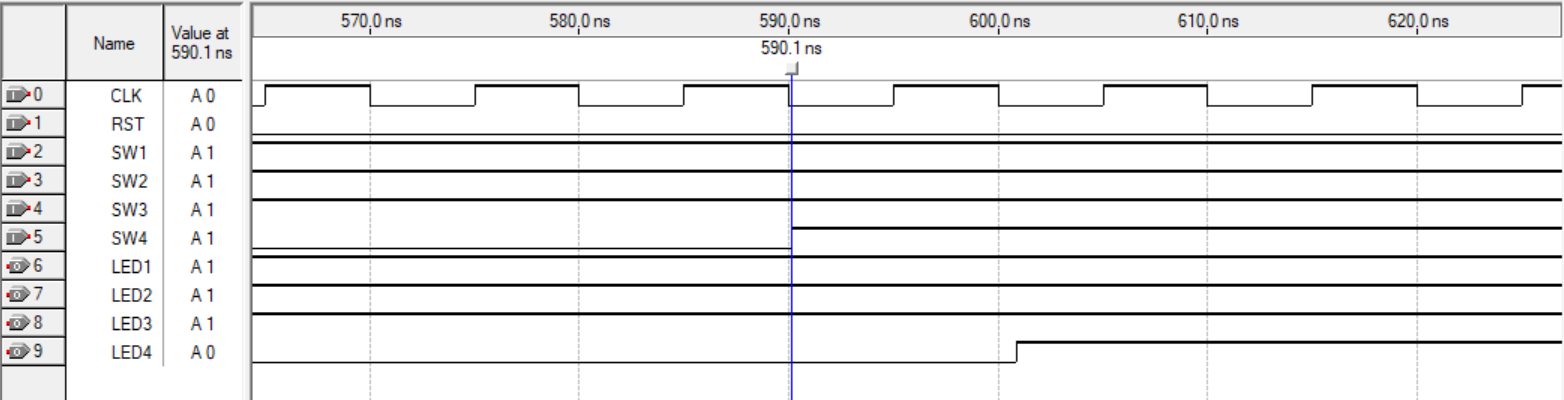


Figure 16 - Timestamp of input transition in the simulation

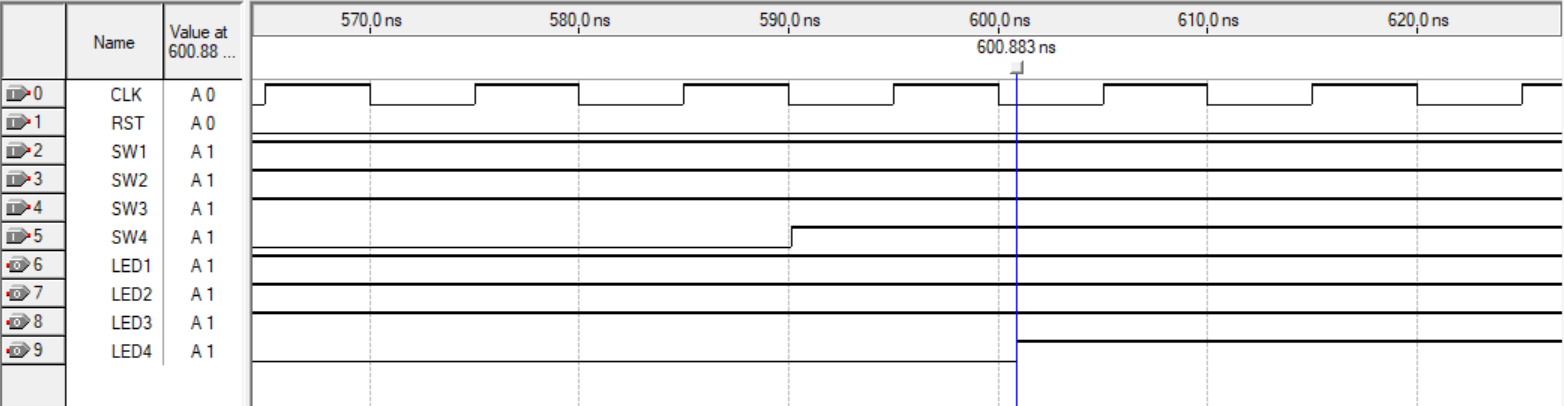


Figure 17 - Timestamp of output transition in the simulation

Next, to quantify this delay, we analyzed the exact moments when the input changed and when the corresponding output transitioned. The first figure captures the moment when the input signal first changes, while the second highlights the exact time the output follows. By examining the timestamps in both figures, we can calculate the propagation delay using the formula:

Propagation Delay = Time at Output Change − Time at Input Change

Propagation Delay = 600.883ns – 590.1ns

Propagation Delay = 10.783ns

This measured delay reflects the time it takes for the signal to travel through the internal flip flops and logic gates before appearing at the output. It confirms that, unlike the ModelSim simulation, real timing behavior introduces small but significant delays that must be accounted for in hardware implementations.

## Power consumption analysis

In our design, various integrated circuits (ICs) contribute to the overall power consumption. To better understand the power requirements, we can break down the consumption for each component based on its static and dynamic power characteristics. Below is a summary of the power consumption for each key IC used in the circuit:

**1. 74LS Series Logic Gates (74LS02, 74LS32, etc.)**

* **Supply Voltage:** Typically 5V (within 2V to 6V range)
* **Static Power:** 1-2μA (negligible)
* **Dynamic Power:** 0.5mA per gate (active)
* **Total Current Draw:** Approximately 5mA for 10 gates operating with a 50pF load at 1MHz.

The 74HC series logic gates, which include AND, OR, and XOR gates, consume very little power at rest (static) and only draw current during active switching. The dynamic power is driven by the charging and discharging of internal capacitive loads, which depends on the number of gates in operation and the switching frequency.

**2. 74LS74 Dual D Flip-Flop**

* **Supply Voltage:** 5V
* **Static Power:** 4μA per flip-flop
* **Dynamic Power:** 0.8mA @ 1MHz, 2.5mA @ 10MHz (clock-dependent)
* **Total Current Draw:** The flip-flop consumes 0.8mA at 1MHz, and 2.5mA at 10MHz.

The 74HC74 dual D flip-flop is used to store and transfer data, and its power consumption varies with the clock frequency. While the static power consumption is quite low, the dynamic power increases with higher clock speeds, making it an essential factor to consider for high-frequency applications.

**3. 555 Timer (Clock Generation)**

* **Supply Voltage:** Typically 5V (within 4.5V to 15V range)
* **Quiescent Current:** 3-10mA (depends on manufacturer)
* **Operating Current:** 10-15mA (when driving outputs)
* **Total Current Draw:** 10-15mA when driving outputs.

The 555 timer generates the clock signal for the circuit. Its quiescent current is low when idle, but the current consumption rises when the timer is actively driving outputs. The current draw is highly dependent on the load and the specific configuration of the timer.

These three ICs form the backbone of the circuit's logic operations and clock generation. The 74LS series logic gates are efficient but draw more current during switching operations, and their dynamic power consumption is directly impacted by the frequency and complexity of the circuit. The 74LS74 flip-flops, while low in static current, consume more power at higher clock frequencies. Lastly, the 555 timer, essential for generating the clock signal, has a noticeable current draw when active.

## Problems faced during the project

One of the most important challenges in this project was designing a system that could correctly identify which switch was turned off last and use that information to activate the corresponding LED sequence. Throughout the development process, we explored several different approaches before arriving at a final solution. Each attempt helped us better understand the behavior of the system and guided us closer to a cleaner and more efficient design.

Initial Attempt: D Flip Flop-Based Detection

Our first idea was to detect falling edges in the switches by using **D flip flops**. The logic behind this approach was to store the current and previous states of each switch. When a switch transitioned from ON to OFF, the system would detect that falling edge and generate a high signal, indicating that this switch was the last one turned off.

A diagram of a computer network

AI-generated content may be incorrect.

Figure 18 - Sequence Detector Circuit Using D Flip Flops (Without SR Latches)

To test this logic, we ran simulations in Quartus. The results showed that the output signal did detect the transition correctly, but only for a very short period. The signal would go high briefly and then immediately drop back to zero. This behavior was not ideal for our system, because we needed the output to remain high and stable until the system was ready to reset and switch to the appropriate LED sequence.

A graph with black lines

AI-generated content may be incorrect.

Figure 19 - Simulation of Circuit Without SR Latches

To fix this problem, we modified the design by adding **SR latches** after the detection logic. The SR latch captured the brief high pulse and held it as a stable output until it was explicitly cleared. This ensured that once a switch was turned off, the corresponding output would remain high and allow the system to detect the correct sequence reliably.

A diagram of a computer

AI-generated content may be incorrect.

Figure 20 - Modified Sequence Detector Circuit Using SR Latches

While this solution worked correctly, it significantly increased the complexity of the circuit. The number of gates and wires made the design bulky and difficult to manage. We realized that for our project goals, this approach was more complicated than necessary and not practical for implementation on a small breadboard setup.

At this point, instead of moving to a simplified direct logic method, we decided to try another formal approach. We attempted to model the system as a **finite state machine**, using **state diagrams** and **state tables** to determine the correct sequence logic.

Second Attempt: State Diagram and Table Method

After moving away from the latch-based design, we tried modeling the system using a finite state machine. Our idea was to define states based on which switches were ON at a given time and use those states to generate the correct output sequence.

We defined the following key states:

* A general state where **more than two switches are ON**, represented as **101**
* A state for **SW1 ON**, represented as **001**
* A state for **SW2 ON**, represented as **010**
* A state for **SW3 ON**, represented as **011**
* A state for **SW4 ON**, represented as **100**
* A special state where **all switches are OFF**, represented as **000**

Our inputs were the four switches:

* **SW1**, **SW2**, **SW3**, **SW4**

The outputs were as follows:

* **Sequence 1**: 00
* **Sequence 2**: 01
* **Sequence 3**: 10
* **Sequence 4**: 11
* **Valid output (V)**: 1 when the input is valid, 0 otherwise

We used this information to draw a full state diagram showing all the transitions and output values.

A diagram of a network

AI-generated content may be incorrect.

Figure 21 - State Diagram for FSM-Based Sequence Detector

Based on the diagram, we constructed a state table listing all current states, inputs, outputs, and transitions.

Table 3- State Table for FSM-Based Sequence Detector

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Inputs | | | | Next State | | | Outputs | | |
| SW1 | SW2 | SW3 | SW4 | E' | H' | L' | SQ1 | SQ2 | V |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | X | X | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | X | X | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | X | X | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | X | X | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | X | X | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | X | X | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | X | X | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | X | X | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | X | X | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | X | X | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | X | X | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | X | X | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | X | X | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | X | X | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | X | X | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | X | X | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | X | X | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | X | X | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | X | X | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | X | X | 0 |

After completing the table, we began deriving Boolean equations for the output logic. However, when simplifying the expressions for selecting the correct sequence, we realized the equations were becoming far too long and complex. Even after removing unnecessary states and grouping similar conditions, the results were not practical for hardware implementation.

A grid with colorful circles and numbers

AI-generated content may be incorrect.

Figure 22 - Example of a Long Output Equation

This complexity suggested that something in our logic or state assignment was incorrect, or at the very least overly complicated. It became clear that this method, while organized, was not suitable for our specific project. The effort required to build and debug such a circuit would have outweighed the benefit of using a structured state machine.

For that reason, we decided not to move forward with the FSM-based design either. While this method helped us explore the problem from a formal perspective and gave us deeper insight into system behavior, it did not meet our need for a simple, reliable, and efficient design.

While the earlier stages of the project presented their own difficulties, the design phase introduced a different set of challenges. Throughout this phase, many parts of the circuit did not behave as expected, and several design decisions led to unforeseen complications. Much of the process was driven by trial and error. The circuit was repeatedly modified and tested in Quartus, with each iteration helping to gradually refine the behavior until the correct output was finally achieved. This iterative approach proved essential in overcoming the complexity of the system and ensuring proper functionality.

## Key design points that present advantages over alternative designs

1. **Modularity and Scalability**: The system is built in a modular way, allowing for easy expansion. Additional switches, LEDs, or sequences can be integrated with minimal changes to the overall structure. This makes the design highly adaptable for future improvements or extensions.
2. **Efficient State Retention**: By incorporating memory elements that store the current state of the system, the circuit maintains functionality even between input changes. This is essential for applications where the system must "remember" previous actions without constantly refreshing the inputs.
3. **Automatic Reset Logic**: The system is designed to detect when all switches are turned off and uses this as a condition to automatically reset itself. This removes the need for external reset buttons and makes the circuit more user-friendly and self-sufficient.
4. **Synchronized Timing Control**: All operations are synchronized using a shared clock signal. This ensures that data is processed in a controlled and predictable manner, which is especially important in systems where timing-related issues can cause instability or incorrect outputs.
5. **Simplified Circuit Complexity**: By relying on standard ICs rather than building everything from basic gates, the overall circuit is less cluttered and more manageable. This reduces the chance of wiring errors, lowers the debugging workload, and enhances the reliability of the system.
6. **Low Gate Count Relative to Functionality**: Despite offering multiple programmable sequences and dynamic input mapping, the design uses a relatively low number of logic gates. This efficiency is achieved by leveraging components that encapsulate complex behavior internally.
7. **User-Responsive Interaction**: The system directly maps switch actions to LEDs in a way that is intuitive for users. Each switch not only turns an LED on but is also responsible for turning it off, enhancing usability and making the design easy to operate without prior instructions.
8. **Dynamic Sequence Detection and Execution**: The circuit can detect which sequence is active based on user interaction and adjust its behavior accordingly. This allows for multiple operational modes without manual reconfiguration or programming changes.
9. **Clean Output Transitions**: Due to the synchronized control and memory-based architecture, transitions between LED states are smooth and free of glitches. This results in a professional and polished output that behaves consistently under different input patterns.

# Conclusion

The Logic Controlled Board project allowed us to apply the digital logic concepts we learned throughout the semester in a practical and hands-on setting. We designed a system that uses finite state machines, sequential logic, and timing circuits to create dynamic behavior based on user interaction. The board responds to the last switch turned off and adjusts the LED activation sequence accordingly, which creates the illusion of intelligent control. Our approach to the design was based on logical thinking and analysis rather than formal truth tables or state diagrams. We focused on understanding how each part of the system should behave and built our solution step by step using clear reasoning. This method allowed us to simplify the design and focus on functionality while still meeting the project requirements. Through this project, we were able to connect theoretical knowledge to real circuit implementation. We gained experience in using Quartus for simulation, assembling the hardware on a breadboard, and solving problems that came up during testing. The project helped us strengthen our understanding of digital systems and gave us the confidence to approach more complex designs in the future.