Hardware Engineering Lab Summer term 2023

Documentation

Team D

12/06/2023

Important Notes:

- ***The descriptions in italics in this document (except for some section headings) are exemplary and explanatory and must be removed from the completed report.***
- ***Identify which section of this report was created by which team member***
- ***Your documentation should have ca. 8 pages (content! Without cover sheet, references, appendix etc.).***

1 Team members

- *Ajay Paul*
- *Doluwamu Taiwo Kuye*
- *MOHAMMAD ASHRAFUZZAMAN SIDDIQI*

2 Introduction

The purpose of this project is to develop an alarm system using the Nexys A7 FPGA as the target device. The alarm system is designed to provide security and protection by monitoring inputs from a keypad and displaying relevant information on a seven-segment display. In addition, the system incorporates a buzzer to generate audible alarms when triggered. The Nexys A7 FPGA serves as the central component for implementing the alarm system's functionality.

Use Case:

The alarm system project has various use cases and scenarios where it can be applied. One primary use case is residential security. The alarm system can be installed in homes to provide an additional layer of protection against unauthorized access. Users can interact with the system using a keypad connected to the Nexys A7 FPGA, entering security codes to arm or disarm the system. When an intrusion is detected, the system activates the buzzer to generate a loud alarm, alerting occupants and deterring intruders.

Another use case is commercial security. The alarm system can be deployed in commercial buildings, offices, or retail stores to safeguard valuable assets and ensure the safety of employees and customers. Authorized personnel can interact with the system through the keypad, entering access codes to control the system's operation. In the event of a security breach, the system triggers the buzzer to emit a loud alarm, attracting attention and prompting immediate response from security personnel.

Additionally, the alarm system can be used in industrial settings where security and access control are critical. It can be integrated into manufacturing facilities, warehouses, or restricted areas to monitor and control access to sensitive zones. The keypad allows authorized personnel to input access codes, and the system validates the codes using the implemented logic on the Nexys A7 FPGA. If unauthorized access is detected, the system activates the buzzer to sound an alarm, alerting nearby personnel and initiating security protocols.

Overall, the alarm system project demonstrates the implementation of a secure and reliable solution using the Nexys A7 FPGA. By integrating a keypad, seven-segment display, and a buzzer, the system provides comprehensive security functionalities. The use cases highlight the system's versatility and applicability in different settings, ensuring robust security measures and timely response to potential threats.

WHY FPGA AND VHDL

FPGAs (Field-Programmable Gate Arrays) and VHDL (Very High-Speed Integrated Circuit Hardware Description Language) play crucial roles in the project domain of developing an alarm system using the Nexys A7 FPGA. Here's why they are important:

1. Flexibility and Customization: FPGAs offer high flexibility and customization capabilities, allowing the design and implementation of complex digital circuits and systems. In the context of the alarm system project, FPGAs enable the integration of various components such as the keypad, seven-segment display, and buzzer into a single programmable device. This flexibility allows for tailored solutions that can meet specific project requirements and easily adapt to changing needs.

2. Hardware Description Language: VHDL is a hardware description language widely used for designing and describing digital circuits. VHDL allows designers to specify the behavior and structure of digital systems at various levels of abstraction. In the alarm system project, VHDL enables the description of the system's functionality, including the interaction between different modules, the logic for keypad input processing, display control, and buzzer activation. VHDL facilitates the translation of design concepts into a format that can be synthesized and implemented on an FPGA.

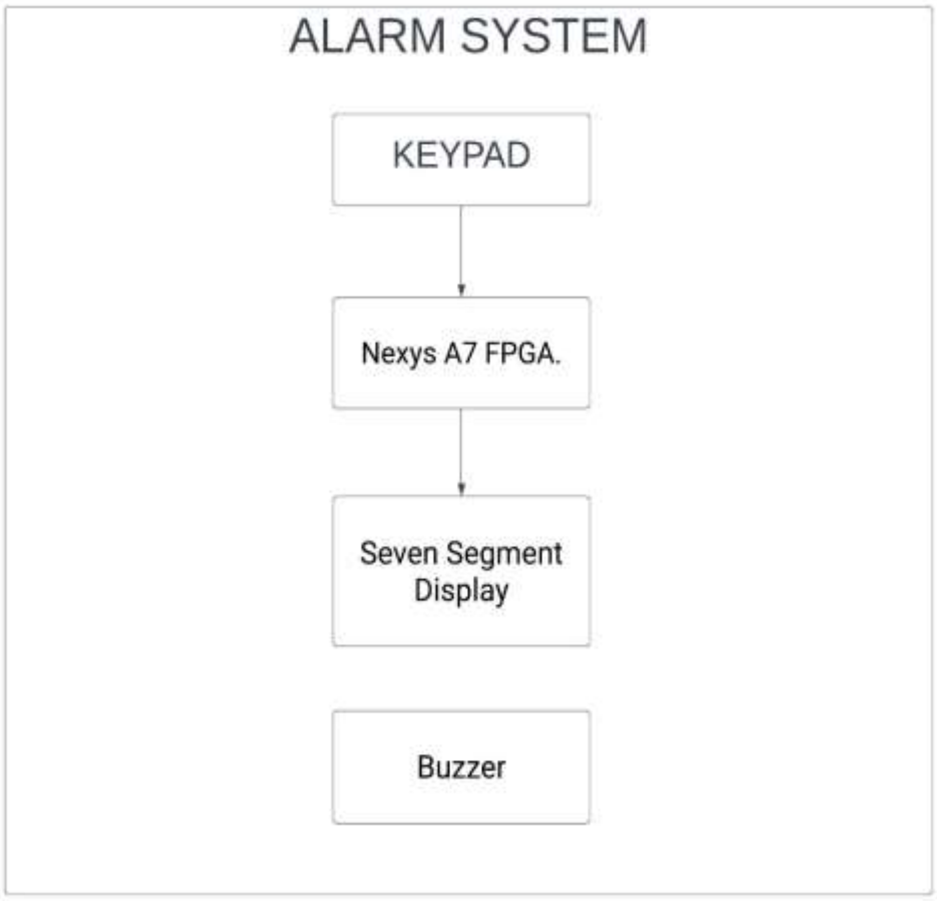
3. Performance and Speed: FPGAs offer high-performance capabilities, enabling the execution of complex algorithms and computations in real-time. In the context of the alarm system, the FPGA's processing power allows for quick and efficient processing of keypad inputs, validation of access codes, and real-time control of the seven-segment display and buzzer. This ensures a responsive and reliable alarm system that can deliver timely security alerts and alarms.

4. Integration and Prototyping: FPGAs provide a platform for integrating multiple hardware components into a single device, reducing the need for external components and simplifying the system's overall design. The Nexys A7 FPGA, specifically, offers a range of I/O interfaces and resources that can be utilized for seamless integration with peripherals such as the keypad and buzzer. This integration capability enables rapid prototyping and testing of the alarm system, allowing for iterative development and refinement.

5. Scalability and Upgradability: FPGAs provide scalability options, allowing for future expansions and upgrades to the alarm system. As the project requirements evolve or new features are desired, FPGAs offer the flexibility to incorporate additional functionalities or modify existing modules without requiring significant hardware changes. This scalability ensures that the alarm system can adapt to future needs and advancements in security technology.

Overall, FPGAs and VHDL are essential in the project domain of developing an alarm system as they provide the necessary tools and capabilities to design, implement, and customize complex digital circuits, enabling high-performance, flexible, and scalable solutions.

3 Concept description



The main application for the prototype is an alarm system. The alarm system is designed to provide security and protection by detecting and responding to unauthorized access or other predefined events. It can be used in various settings such as residential buildings, offices, commercial establishments, or any other location where security is a concern.

The prototype incorporates features such as a keypad for user input, an FPGA for processing and controlling system operations, a display for showing system status and feedback, and a buzzer for generating audible alarms. The system allows users to enter access codes to arm or disarm the alarm, and it triggers alarms if unauthorized access is detected or if certain predefined conditions are met.

The alarm system prototype aims to provide a reliable and effective solution for enhancing security and peace of mind in various environments. Its flexibility allows for customization and scalability to meet specific requirements and integrate with other security systems if needed.

4 Project/Team management

Our project management style was the Adaptive or Flexible Management approach:
This allows for flexibility and adaptation to changing project requirements and circumstances. We structured task according to areas of specialisation and interest.
This Encourages frequent communication, collaboration, and quick decision-making.
Ideally, we all contributed to every aspect but due to unlimited hardware we took turns in implementing different parts of the system, we also made sure that every member of the group understood what was being done in each level of development.

Tasks management

The main tasks involved:

1. *Vhdl modules code*
2. *Test bench*
3. *Schematics design*
4. *PCB Design*

Doluwamu Taiwo Kuye

- *VHDL(1key,2key,4keys) all modules*
- *SCHEMATICS*
  - *FPGA component design*
  - *Seven segment design*
  - *Power design*
- *Place and Route*
- *Concept Description*
- *Testing of VHDL code on Fpga*
- *3d pcb view*
- *Documentation of High-level architecture*
- *Documentation of VHDL Modules*

MOHAMMAD ASHRAFUZZAMAN SIDDIQI

VHDL Test bench (for all modules)
Concept description
Place and route for design 2
VHDL 2 Key implementation(comparator, latch ,fsm modules).

Ajay Paul

- First version VHDL - Comparator, Latch
- Testbench for first version VHDL - Comparator, Latch
- Schematics of Full system

1. *Voltage regulation for the FPGA*
2. *Power to FPGA, Oscillator, Reset and grounding*
3. *Seven Segment module implementation for FPGA*
4. *Implementation of external keys and LEDs for FPGA*
5. *FPGA JTAG Interface and Pin header for Xilinx programmer*

- PCB Layout
- 3D PCB view
- PCB design schematic summary

5 Technologies

To implement the project, the following technological approaches will be utilized:

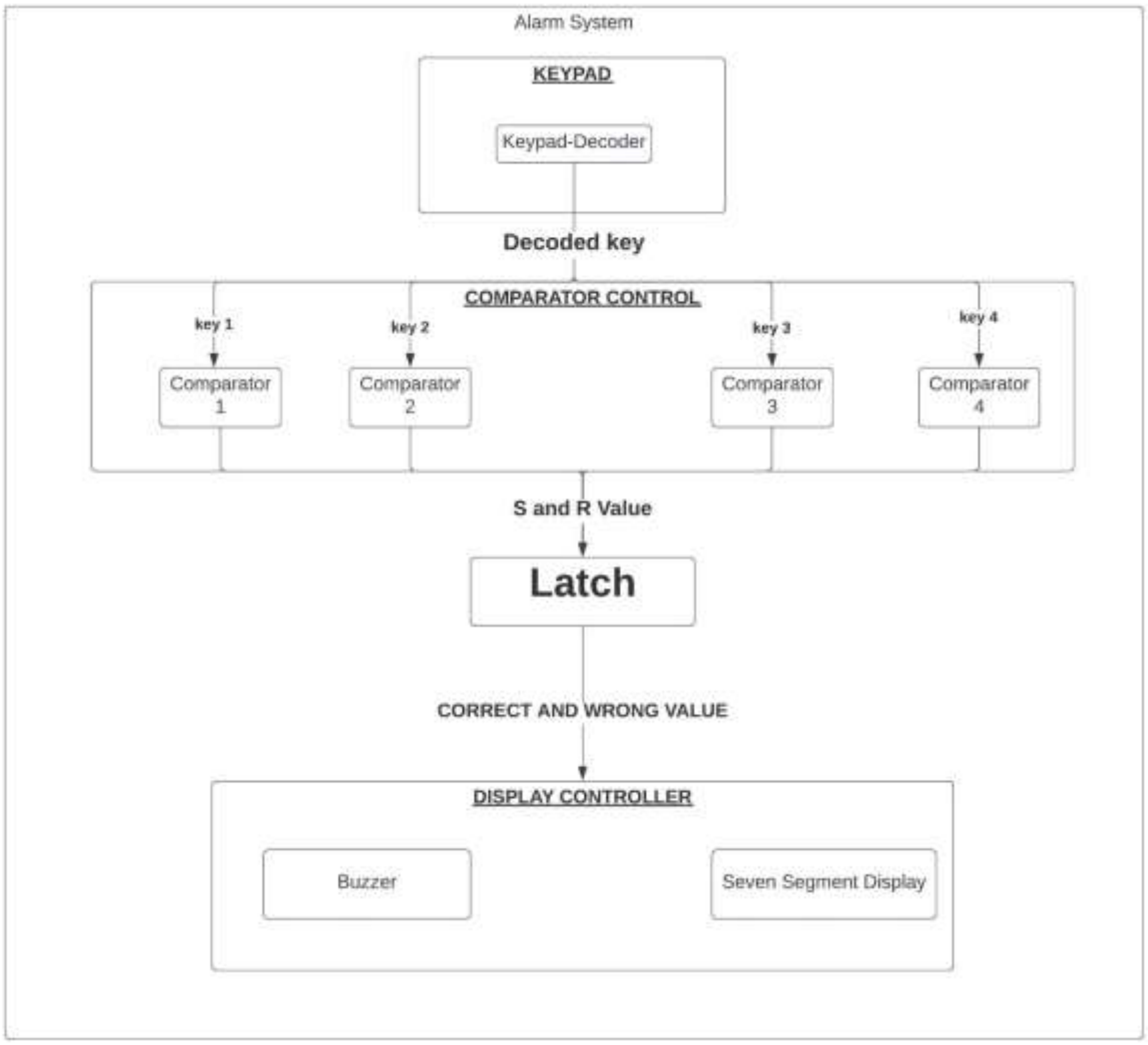
1**. VHDL (Very High-Speed Integrated Circuit Hardware Description Language):**
VHDL will be used as the hardware description language to design and describe the functionality of the digital circuits and components within the project. VHDL provides a standardized and structured approach to designing hardware, allowing for efficient and reliable implementation of complex digital systems.

2. **FPGA (Field-Programmable Gate Array):**
The target device for the project is the Nexys A7 FPGA. FPGAs are programmable integrated circuits that can be configured to implement custom digital circuits and systems. The FPGA will serve as the main processing unit, responsible for executing the logic defined in VHDL and controlling the overall operation of the alarm system.

3. **KiCAD:**
KiCAD is an open-source software suite used for electronic design automation (EDA). It provides a set of tools for schematic capture, PCB layout design, and component library management. KiCAD will be utilized to design the circuit schematics and create the PCB layout for the alarm system. It enables the placement and routing of components, ensuring proper connectivity and signal integrity.

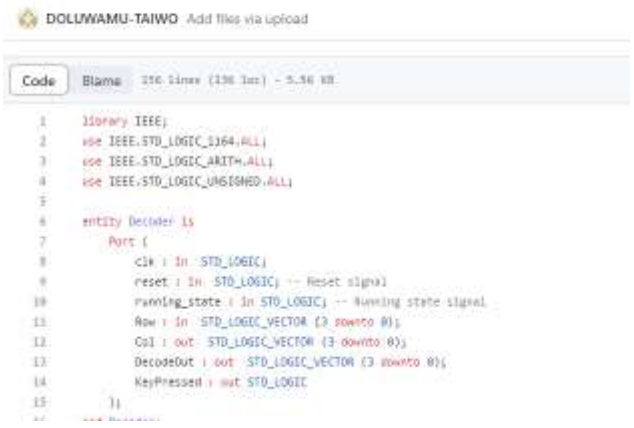
By combining VHDL for digital design, FPGA for hardware implementation, and KiCAD for PCB design, the project can be efficiently realized. These technological approaches provide a comprehensive solution for developing and deploying the alarm system prototype, enabling customization, flexibility, and reliable performance.

6 VHDL and FPGA Implementation



The digital design implementation involves several interconnected modules to perform the desired functionality. The input module (keypad decoder) receives input signals and may include signal conditioning or preprocessing logic. There is also a finite state machine that controls the decoded-out values; this is for making sure the keys pressed are sequential and assigned to the right comparator module. The heart of the design lies in the four comparator modules, which compare pairs of input signals and generate output signals indicating the result of the comparison (equal or not). These comparators can be instantiated as separate components in the design, with their inputs connected to the appropriate signals from the input module. The Output Module (Display Controller) handles the output signals generated by the comparators and the latch; it includes ports to transmit the comparison results to other parts of the system or display them. The interconnect between these modules involves routing the input signals from the input module to the comparators and connecting the output signals from the comparators to the output module. Several other finite state machine modules exist to ensure the correct interconnection of signals, enabling proper data flow and functionality in the design.

DECODER MODULE


DOLUWAMU-TAIWO Add files via upload
Code
Blame
156 lines (136 loc) · 5.56 KB

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

entity Decoder is
    Port (
        clk : in  STD_LOGIC;
        reset : in  STD_LOGIC; -- Reset signal
        running_state : in STD_LOGIC; -- Running state signal
        Row : in  STD_LOGIC_VECTOR (3 downto 0);
        Col : out  STD_LOGIC_VECTOR (3 downto 0);
        DecodeOut : out  STD_LOGIC_VECTOR (3 downto 0);
        KeyPressed : out STD_LOGIC
    );
```

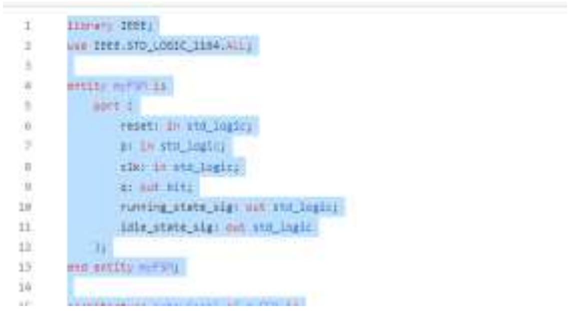
The "Decoder" module in VHDL is responsible for decoding the inputs from the keypad and determining the corresponding key that was pressed. Here is the explanation of the code:

- *The entity "Decoder" declares the input and output ports of the module, including the clock signal (clk), reset signal (reset), running state signal (running_state), input row signal (Row), output column signal (Col), decoded output signal (DecodeOut), and a key pressed indicator signal (KeyPressed).*
- *In the architecture "Behavioral", the signals sclk (used for clock synchronization) and AnyKeyPressed (indicator for any key press) are declared.*
- *Inside the process block, the module responds to changes in the clock (clk), reset, and running state signals.*
- *If the reset signal is asserted (reset = '1'), the module resets the relevant signals to their initial values.*
- *During normal operation (when clk'event and clk = '1' and running_state = '1'), the module performs the following actions:*

- *It checks the value of the signal sclk to determine the timing of different operations.*
- *For each specific timing interval, it checks the row inputs (Row) to determine which key was pressed.*
- *Based on the row inputs, it assigns the corresponding column value (Col) and the decoded output value (DecodeOut) for the pressed key.*
- *It sets the AnyKeyPressed signal to '1' if any key is pressed during the timing interval.*
- *The sclk signal is incremented according to the specified timing intervals.*
- *Finally, the AnyKeyPressed signal is assigned to the KeyPressed output, indicating whether any key was pressed.*

This Decoder module serves as a crucial component in the overall functionality of the system, enabling the identification and decoding of key presses from the keypad, which are then utilized for further processing in the system.

<u>MyFSM Module</u>

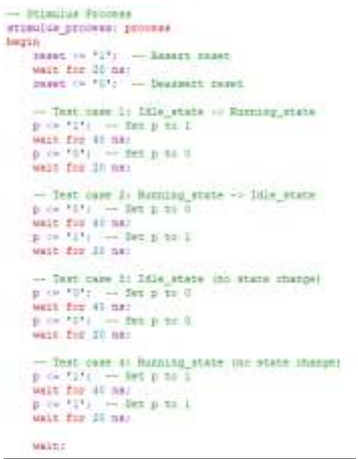


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity myFSM is
    port (
        reset: in std_logic;
        p: in std_logic;
        clk: in std_logic;
        q: out bit;
        running_state_sig: out std_logic;
        idle_state_sig: out std_logic
    );
end entity myFSM;

[illegible]
```

The "myFSM" module in VHDL is a finite state machine responsible for controlling the state transitions based on the input signal "p" and the clock signal "clk." It features two states, "Idle_state" and "Running_state," and provides outputs to indicate the current state, "running_state_sig" and "idle_state_sig," as well as the output signal "q" that reflects the state.
The module consists of two processes:
1. "state_memory" process updates the current state based on the clock and reset signals. It initializes the current state to "Idle_state" when the reset signal is asserted and updates it to the next state on the rising edge of the clock signal.
2. "FSM_control" process determines the next state based on the current state and the input signal "p." It uses a case statement to define the state transition logic, transitioning to the "Running_state" when "p" is asserted and returning to the "Idle_state" when "p" is deasserted during the "Running_state."
The output signals "running_state_sig" and "idle_state_sig" are assigned based on the current state, indicating whether the FSM is in the "Running_state" or "Idle_state," respectively. The output signal "q" reflects the state, being '1' when the FSM is in the "Running_state" and '0' otherwise.

The "myFSM" module plays a vital role in the overall system, providing the necessary control and state management for efficient operation based on the input signal "p" and clock signal "clk."

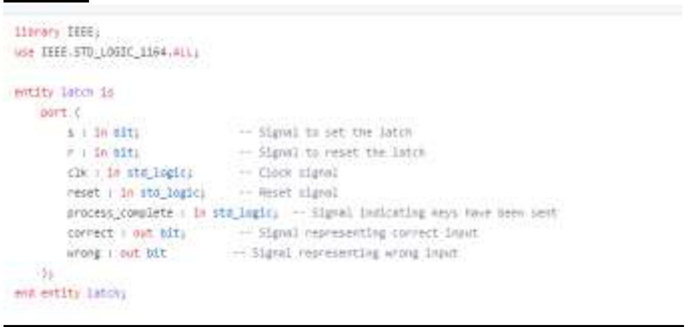
<u>TestBench (MyFSM)</u>



```
-- Stimulus Process
stimulus_process: process
begin
    reset <= '1';  -- Assert reset
    wait for 20 ns;
    reset <= '0';  -- Deassert reset

    -- Test case 1: Idle_state -> Running_state
    p <= '1';  -- Set p to 1
    wait for 45 ns;
    p <= '0';  -- Set p to 0
    wait for 10 ns;

    -- Test case 2: Running_state -> Idle_state
    p <= '1';  -- Set p to 0
    wait for 40 ns;
    p <= '1';  -- Set p to 1
    wait for 20 ns;

    -- Test case 3: Idle_state (no state change)
    p <= '0';  -- Set p to 0
    wait for 45 ns;
    p <= '0';  -- Set p to 0
    wait for 20 ns;

    -- Test case 4: Running_state (no state change)
    p <= '1';  -- Set p to 1
    wait for 40 ns;
    p <= '1';  -- Set p to 1
    wait for 20 ns;

    wait;
```

The test bench myFSM_tb is designed to verify the functionality of the myFSM entity. It stimulates the inputs of the myFSM entity and monitors its outputs to check if the state transitions occur correctly.
Idle_state to Running_state Transition:
Start from the reset state.
Transition from Idle_state to Running_state by setting p to '1'.
Maintain the Running_state by keeping p as '1'.
Running_state to Idle_state Transition:
Transition from Running_state to Idle_state by setting p to '0'.
Maintain the Idle_state by keeping p as '0'.
Idle_state (No State Change):
Remain in the Idle_state by keeping p as '0'.
Running_state (No State Change):
Remain in the Running_state by keeping p as '1'.
These test cases cover different state transitions and also include scenarios where no state change occurs. They aim to verify the correctness of the state transition logic implemented in the myFSM entity.

LATCH



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

entity latch is
    port (
        s : in bit;                -- Signal to set the latch
        r : in bit;                -- Signal to reset the latch
        clk : in std_logic;        -- Clock signal
        reset : in std_logic;      -- Reset signal
        process_complete : in std_logic;  -- Signal indicating keys have been sent
        correct : out bit;         -- Signal representing correct input
        wrong : out bit           -- Signal representing wrong input
    );
end entity latch;
```

The "latch" module in VHDL implements a latch with set and reset functionality. It features inputs for the set signal "s," reset signal "r," clock signal "clk," reset signal "reset," and a signal "process_complete" indicating when keys have been sent. The outputs include signals "correct" and "wrong" representing correct and wrong inputs, respectively.

- *Inside the "behavioral" architecture, the module uses a process sensitive to changes in the clock, reset, and process_complete signals. When the reset signal is asserted, the latch is reset, and the output signals "correct" and "wrong" are cleared.*
- *On the rising edge of the clock signal, if the process_complete signal is asserted, the latch_active signal is set to indicate that the latch process is active.*

- *When the latch is active, the module checks the values of the set signal "s" and the reset signal "r." If "s" is asserted and "r" is deasserted, the output signal "correct" is set to indicate a correct input, and "wrong" is cleared. For all other cases, "correct" is cleared, and "wrong" is set.*
- *If the latch is not active, both "correct" and "wrong" signals are cleared.*

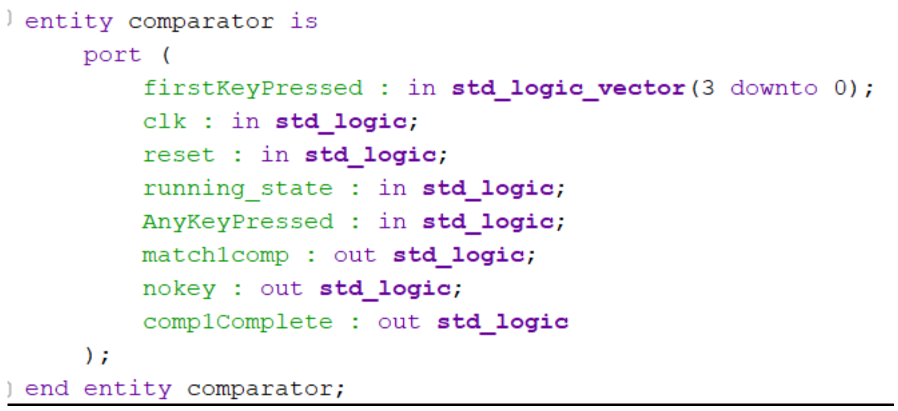
The "latch" module serves as a crucial component in the system, providing the ability to capture and determine the correctness of inputs based on the set and reset signals, ensuring the correct functioning of the overall system.

TestBench(LATCH)

Input Combinations: Different input combinations are applied to the latch module while the process_complete signal is asserted. This includes changing the values of s and r to test various scenarios. Specifically, the following combinations are tested:
When latch_active = '0': s = '0', r = '0'
When latch_active = '1': s = '1', r = '0'
When latch_active = '1': s = '0', r = '1'
When latch_active = '1': s = '1', r = '1'
Each input combination is applied for a duration of 10 ns, allowing the latch module to respond and update its outputs accordingly.

End of Simulation: The simulation continues indefinitely after the last input change, and the testbench waits indefinitely.
By following this predefined test logic, the testbench evaluates the behavior of the latch module and observes the values of the output signals (correct and wrong) throughout the simulation.

COMPARATOR



```
entity comparator is
    port (
        firstKeyPressed : in std_logic_vector(3 downto 0);
        clk : in std_logic;
        reset : in std_logic;
        running_state : in std_logic;
        AnyKeyPressed : in std_logic;
        match1comp : out std_logic;
        nokey : out std_logic;
        comp1Complete : out std_logic
    );
end entity comparator;
```

The "comparator" module in VHDL is designed to compare the first key pressed with a predefined password. It features inputs such as "firstKeyPressed" for the input of the first key pressed, "clk" for the clock signal, "reset" for the reset signal, "running_state" for the running state signal from the FSM, and "AnyKeyPressed" indicating if any key is currently pressed. The module provides outputs including "match1comp" indicating a match with the first password, "nokey" indicating that no key is currently pressed, and "comp1Complete" indicating the completion of the comparison process for "comp1".

Inside the "behavioral" architecture, the module uses a process sensitive to changes in the clock, reset, "firstKeyPressed", "running_state", and "AnyKeyPressed". When the reset signal is asserted, the outputs "nokey", "match1comp", and "comp1Complete" are cleared.

On the rising edge of the clock signal, the module checks the value of "AnyKeyPressed". If no key is currently pressed, the output "nokey" is set to indicate this condition, while "match1comp" and "comp1Complete" are cleared.

If the system is in the running state (indicated by "running_state" being asserted), the module compares the "firstKeyPressed" with the predefined password. If there is a match, the output "match1comp" is set to indicate the match, and "comp1Complete" is set to indicate the completion of the comparison process for "comp1". Otherwise, "match1comp" is cleared.

If the system is not in the running state, all outputs are cleared, as there is no operation during this state.

The "comparator" module is a vital component in the system as it enables the comparison of the first key pressed with the password. It provides feedback regarding the correctness of the input, ensures proper system functionality, and contributes to the overall security of the system.

<u>TESTBENCH</u>

The test bench module "comparator_tb" is implemented to verify the functionality of the "comparator" module. It uses a clock signal and various input signals to stimulate the "comparator" module and observe its outputs. The test bench follows a behavioral architecture.

Inside the architecture, the "comparator" module is instantiated as a component. It has the same port mapping as the original module.

The test bench includes signal declarations for the clock signal ("clk"), reset signal ("reset"), running state signal ("running_state"), any key pressed signal ("AnyKeyPressed"), and the first key pressed signal ("firstKeyPressed"). It also declares signals for the outputs of the "comparator" module: "match1comp", "nokey", and "comp1Complete".

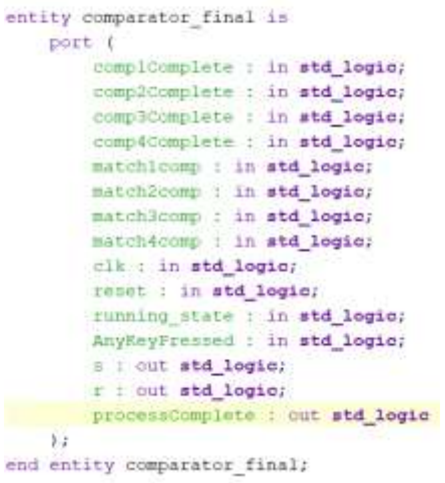
The stimulus process applies various test cases to the inputs of the "comparator" module. It starts by setting the reset signal to '1' and the running state signal to '1' to ensure the initial state. Then it tests the following scenarios:

1. ***<u>Test case 1</u>**: No key is pressed. It asserts the reset signal and sets the AnyKeyPressed signal to '0'. This tests the behavior when no key is pressed.*
2. ***<u>Test case 2</u>**: Key is pressed but does not match. It deasserts the reset signal, sets the AnyKeyPressed signal to '1', and sets a non-matching value for the firstKeyPressed signal. This tests the behavior when a key is pressed but does not match the password.*
3. ***<u>Test case 3</u>**: Key matches. It changes the value of the firstKeyPressed signal to match the password. This tests the behavior when the firstKeyPressed matches the password.*
4. ***<u>Test case 4</u>**: No key is pressed again, and the running state is '0'. It sets the firstKeyPressed signal to '0000'. This tests the behavior when no key is pressed and the running state is not active.*

The test bench module verifies the functionality of the "comparator" module by stimulating it with different test cases and observing the outputs. It provides a comprehensive test environment to assess the correctness of the "comparator" module's behavior and its interaction with the input signals.

<u>NOTE: There are 3 more exact comparators like this checking the 3 other keys, they have output signals that vary from each other. The test bench can be used for every other comparator.</u>

<u>COMPARATOR FINAL</u>



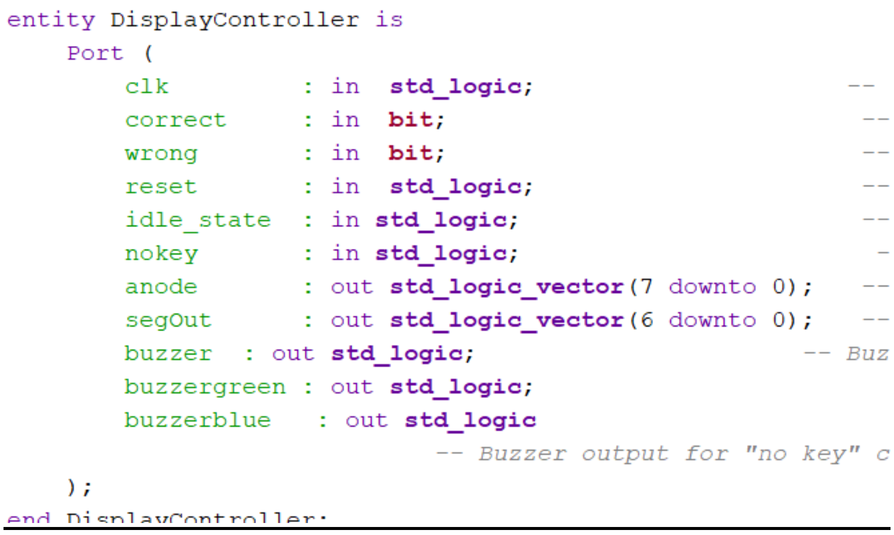
```
entity comparator_final is
    port (
        comp1Complete : in std_logic;
        comp2Complete : in std_logic;
        comp3Complete : in std_logic;
        comp4Complete : in std_logic;
        match1comp : in std_logic;
        match2comp : in std_logic;
        match3comp : in std_logic;
        match4comp : in std_logic;
        clk : in std_logic;
        reset : in std_logic;
        running_state : in std_logic;
        AnyKeyPressed : in std_logic;
        s : out std_logic;
        r : out std_logic;
        processComplete : out std_logic
    );
end entity comparator_final;
```

The "comparator_final" module compares multiple passwords match signal and determines their success or failure. It takes inputs for completion signals (comp1Complete, comp2Complete, comp3Complete, comp4Complete) and match signals (match1comp, match2comp, match3comp, match4comp). The module operates on the rising edge of the clock signal, considering the running state and whether any key is pressed. It sets the outputs (s, r, processComplete) based on the completion and match signals, indicating the success or failure of the password comparisons and the completion status of the process.

<u>Comparator final Testbench</u>

For the comparator final test, the idea was to vary the input signals match1comp, match2comp, match3comp and match4comp, we set these signals to 1 and 0's and monitored the output results. Running state was also set to 1 to enable the test to produce outputs.

Display Controller



```
entity DisplayController is
    Port (
        clk          : in  std_logic;                     --
        correct      : in  bit;                            --
        wrong        : in  bit;                            --
        reset        : in  std_logic;                      --
        idle_state   : in std_logic;                       --
        nokey        : in std_logic;                        -
        anode        : out std_logic_vector(7 downto 0);   --
        segOut       : out std_logic_vector(6 downto 0);   --
        buzzer    : out std_logic;                     -- Buz
        buzzergreen : out std_logic;
        buzzerblue    : out std_logic
                                    -- Buzzer output for "no key" c
    );
end DisplayController;
```

The "DisplayController" module controls a display and buzzer based on input signals. It takes inputs for correct, wrong, reset, idle_state, and nokey. The module operates on the rising edge of the clock signal. It sets the outputs (anode, segOut, buzzer, buzzergreen, buzzerblue) based on the input conditions.

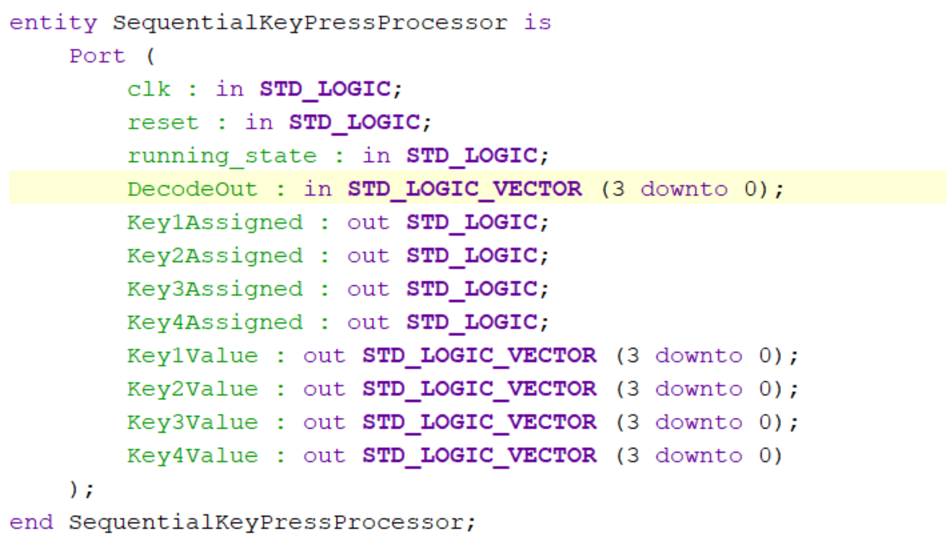
The anode output is set to "11111110" to display only the leftmost digit. The currentDisplay signal represents the value to be displayed. The buzzerStateMain, buzzerStateblue, and buzzerstategreen signals control the states of the main buzzer, "no key" buzzer, and the green LED, respectively.

The process inside the module checks the input conditions and updates the currentDisplay and buzzer state accordingly. When the reset signal is active, the currentDisplay is set to display zero, and both buzzers are turned off. In the idle state, the currentDisplay displays "1", and both buzzers are turned off. When no key is pressed, the currentDisplay displays "0", and the "no key" buzzer is turned on. For correct input, the currentDisplay displays "C", and both buzzers are turned off. For wrong input, the currentDisplay displays "F", and the main buzzer is turned on. In all other cases, the currentDisplay is turned off, and all buzzers are turned off.
The outputs are assigned their respective values based on the current states.
The segOut output is assigned the value of currentDisplay.
The buzzer output is assigned the value of buzzerStateMain.
The buzzerblue output is assigned the value of buzzerStateblue.
The buzzergreen output is assigned the value of buzzerstategreen.

Display Controller Test Bench

The testbench includes two processes:
"clk_process" is responsible for generating the clock signal. It toggles the clock signal every 10 ns.
"stimulus" is the main stimulus process that drives the input signals of the "DisplayController" module. It sets the initial values of the input signals and then provides specific test cases by changing the values of the input signals at different time intervals. The test cases include setting the values for correct_in, wrong_in, reset_in, idle_state_in, and nokey_in.
The testbench waits indefinitely at the end, allowing the simulation to continue until manually stopped.
This testbench allows you to simulate and observe the behavior of the "DisplayController" module under different test scenarios.

SequentialKeyPressProcessor



```
entity SequentialKeyPressProcessor is
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        running_state : in STD_LOGIC;
        DecodeOut : in STD_LOGIC_VECTOR (3 downto 0);
        Key1Assigned : out STD_LOGIC;
        Key2Assigned : out STD_LOGIC;
        Key3Assigned : out STD_LOGIC;
        Key4Assigned : out STD_LOGIC;
        Key1Value : out STD_LOGIC_VECTOR (3 downto 0);
        Key2Value : out STD_LOGIC_VECTOR (3 downto 0);
        Key3Value : out STD_LOGIC_VECTOR (3 downto 0);
        Key4Value : out STD_LOGIC_VECTOR (3 downto 0)
    );
end SequentialKeyPressProcessor;
```

The "SequentialKeyPressProcessor" module is responsible for processing sequential key presses. It takes the clock signal, reset signal, running state signal, and the decoded output of the key presses as inputs. It provides outputs indicating whether each key has been assigned and the values of the assigned keys.
The architecture of the module, "Behavioral," includes signal declarations for the previous key value and the values and assignment status of the four keys.

Inside the process sensitive to the clock signal, the module performs the following actions:

- When the reset signal is active (reset = '1'), all signals are reset to their initial values.
- When the running state signal is active (running_state = '1'), the module checks if the decoded output (DecodeOut) is different from the previous key value (prev_key). If they are different, it assigns the value to an unassigned key (if available) and updates the assigned status accordingly.
- The previous key value (prev_key) is updated with the current decoded output.

Finally, the outputs of the module (Key1Assigned, Key2Assigned, Key3Assigned, Key4Assigned, Key1Value, Key2Value, Key3Value, Key4Value) are assigned the corresponding internal signal values.

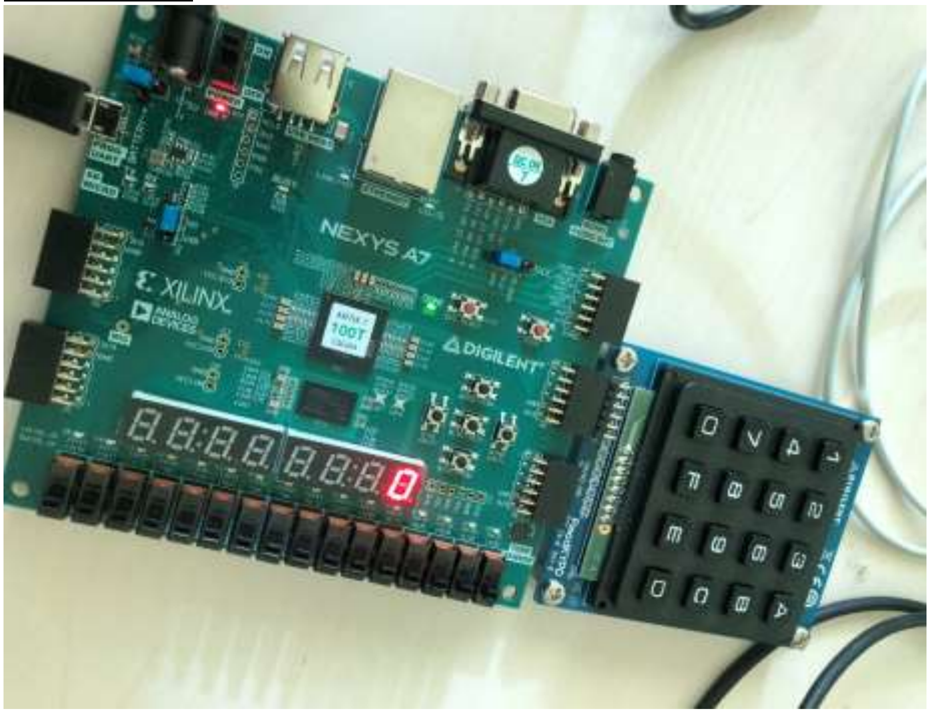
The module effectively tracks and assigns sequential key presses to the available keys based on the running state and the decoded outputs.

SequentialKeyPressProcessor(TestBench)

The testbench sets running_state to '1' and passes four different decoded keys (0001, 0010, 0100, 1000) to the SequentialKeyPressProcessor module. It waits for a duration of 10 ns after setting each key before moving on to the next one. This allows us to monitor if the decoded keys are being assigned sequentially and to the right key pressed holder.

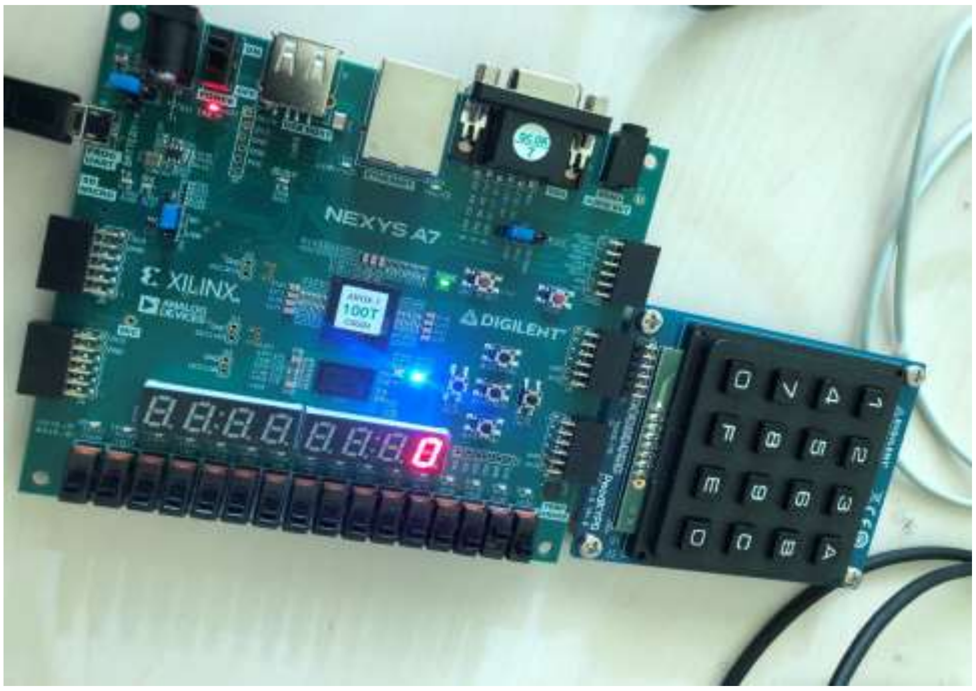
RESULTS SUMMARY

Idle State



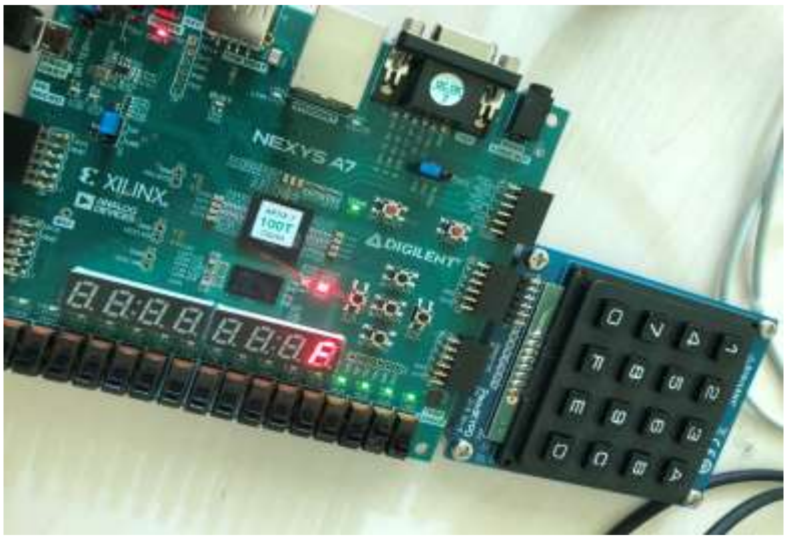
When we Generate the bitstream and program the fpga the system starts in this idle state where it displays 0 this is because of a condition from our display controller where if it has not signal from p (transition enabler from idle to running state) ,nokey or correct and wrong the seven segment anode is set to zero.

RUNNING STATE (NO KEY =1)



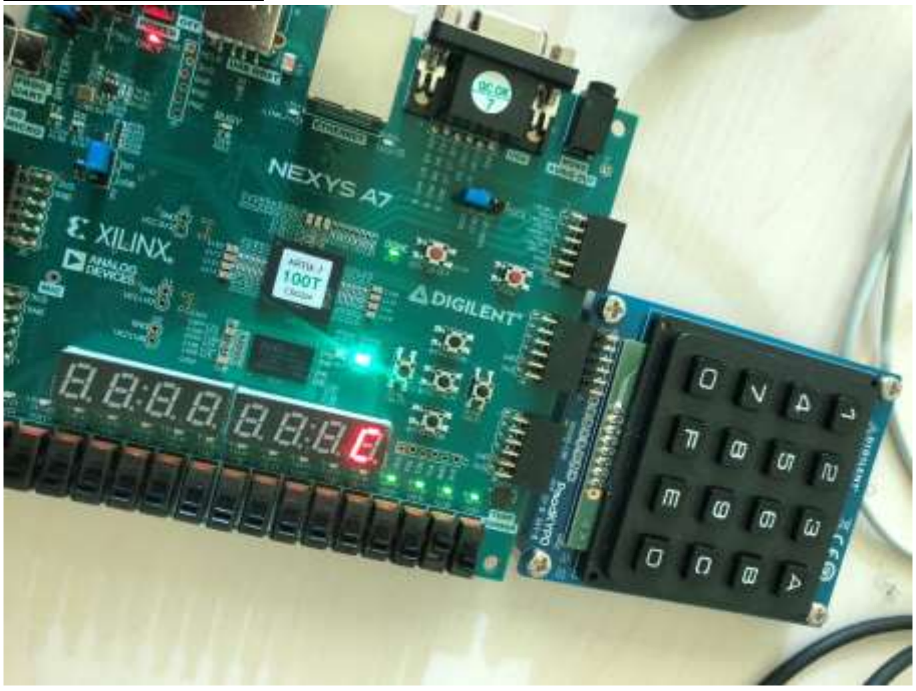
In this State we can notice that the L16 pin is activated, this transitions the system into the running state and also the no key state where it waits for a key press. In this states the led turns blue.

<u>FAILED STATE</u>



If 4 keys are pressed and they are not the same with the set 4 keys this will result in a failed state. The system can now be reset by toggling the J15 (reset) switch, to try again. In this state the seven segment displays F and the Led turns Red.

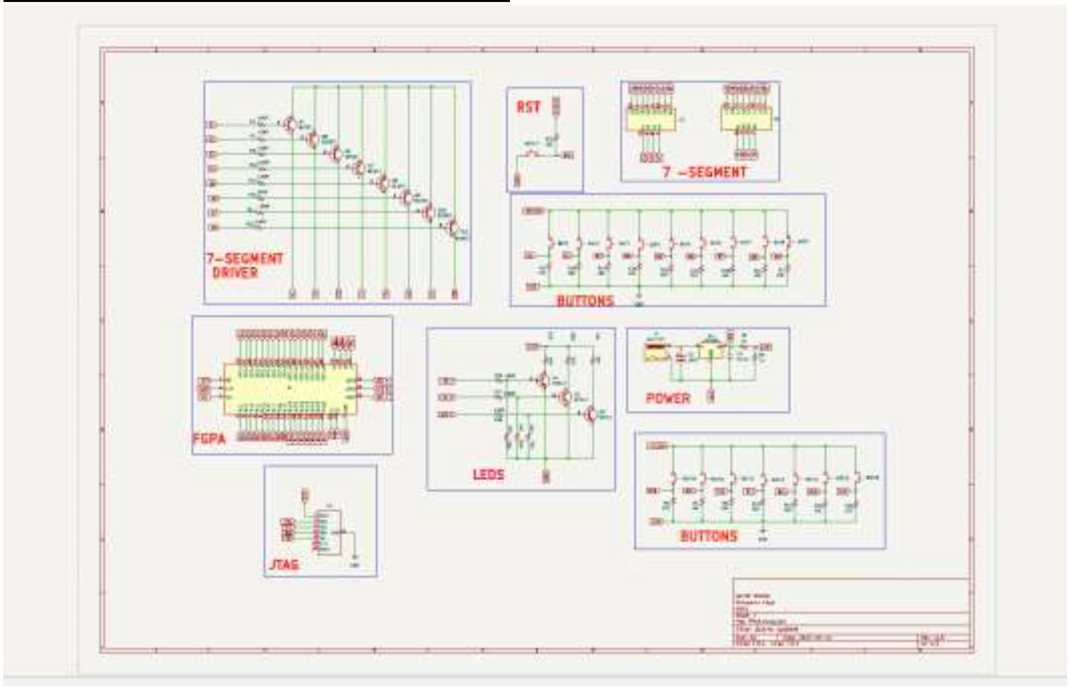
Correct State



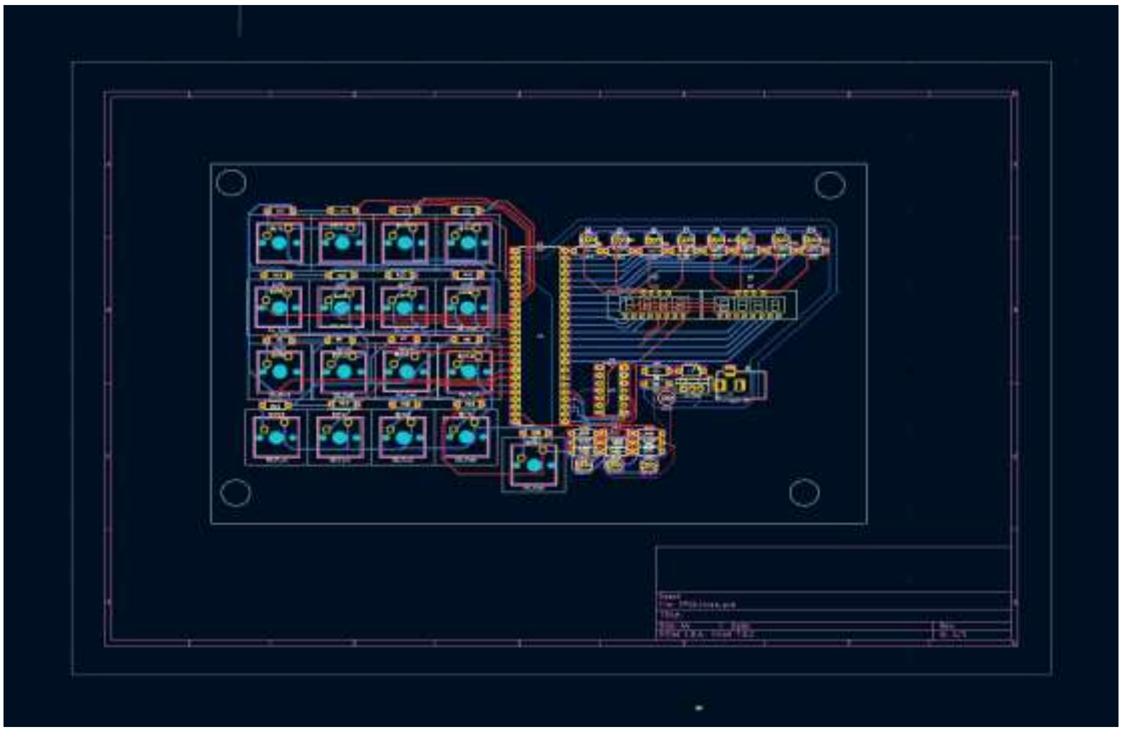
If all the keys matches with the previously set keys we enter into the correct condition, where the LED displays green and the seven segment displays C , it is also good to note that for ever key pressed we have leds that turn on to signify the keys pressed. The reset button can be used to reset the whole system and type in another password.

7 PCB Design

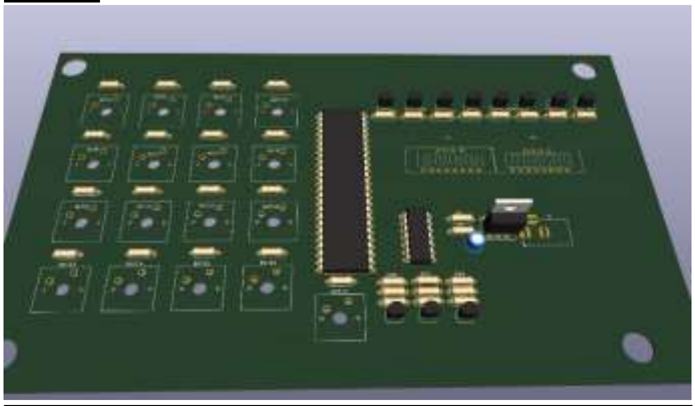
Schematics Design 1(Doluwamu)



PCB LAYOUT AND ROUTING DESIGN 1(Ashraf)



3dview

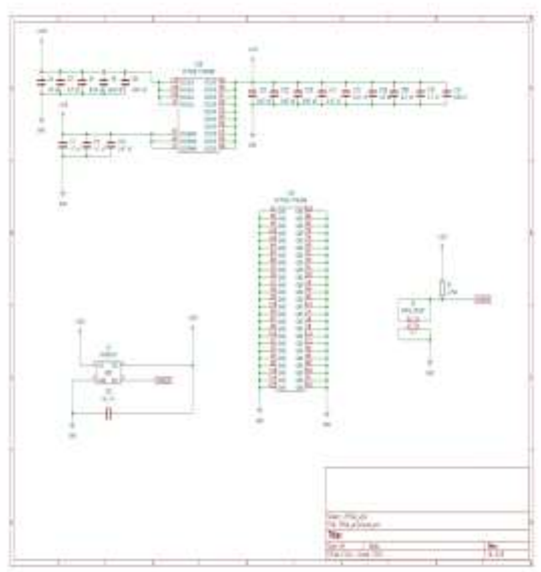


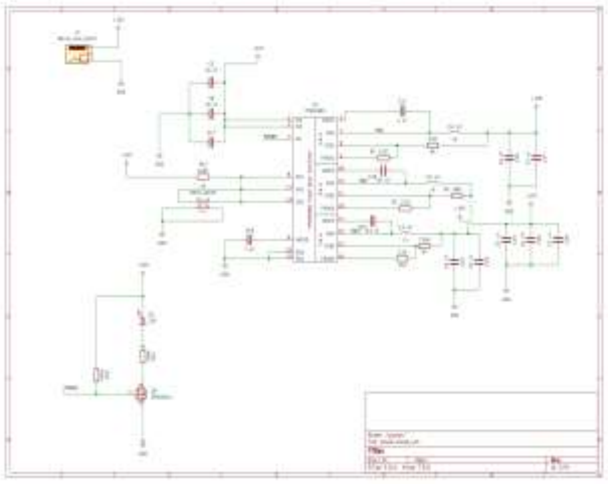
Describe the implementation of your schematic and PCB design.
Give a summary about your PCB design results (Layout, BOM, Costs, Size, etc.)

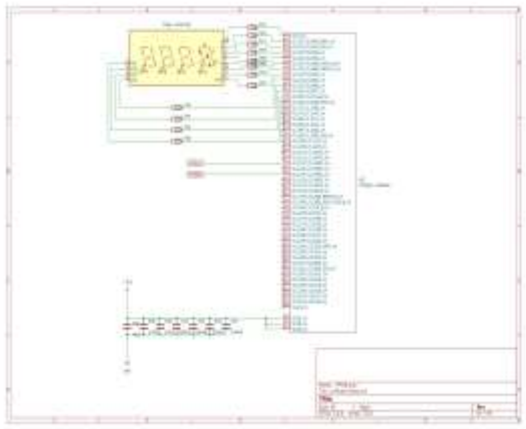
DESIGN 2 Schematics (Ajay)

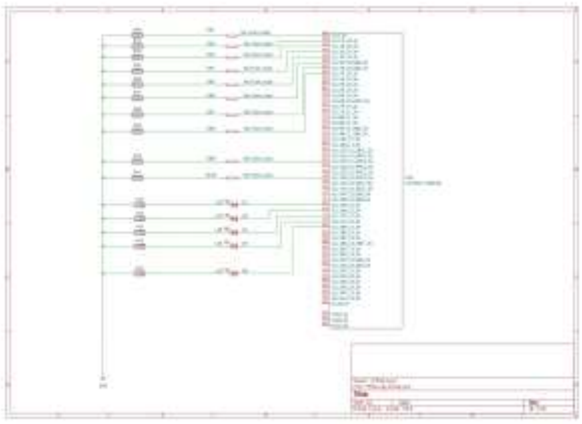
PCB schematic and layouts are implemented using KiCad. Here used Spartan 7 FPGA XC7S50-1FTG196 as the FPGA and A seven segment module, push buttons and LEDs are used as external devices. FPGA XC7S50-1FTG196 is set up with the desired voltage and clock. For this, a voltage regulator is implemented using TPS65581 with output of 1.0V, 1.8V and 3.3V. The clock signal is implemented using an oscillator. As well as, FPGA JTGA interface and pin header for Xilinx JTAG programmer.

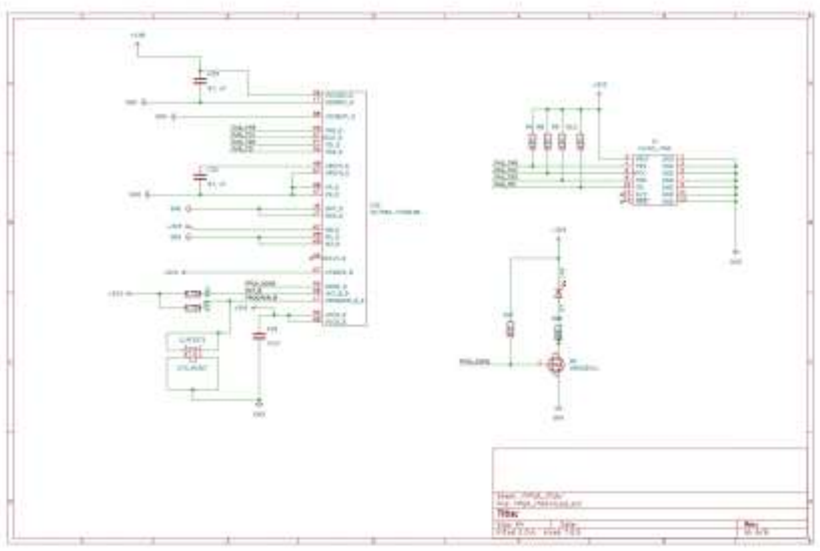
The following schematic are in the order of
1. Voltage regulation for the FPGA
2. Power to FPGA, Oscillator, Reset and grounding
3. Seven Segment module implementation for FPGA
4. Implementation of external keys and LEDs for FPGA
5. FPGA JTAG Interface and Pin header for Xilinx programmer



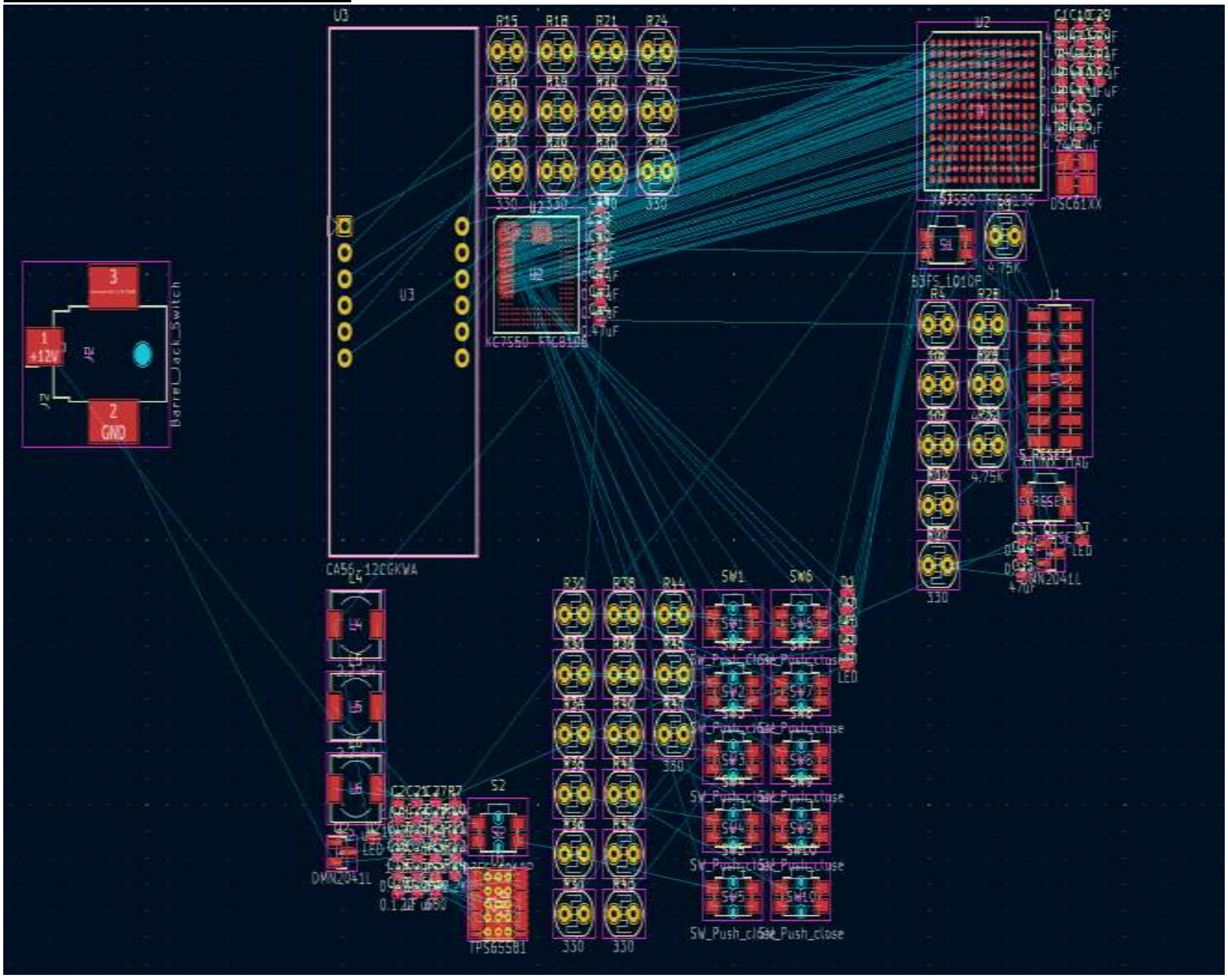




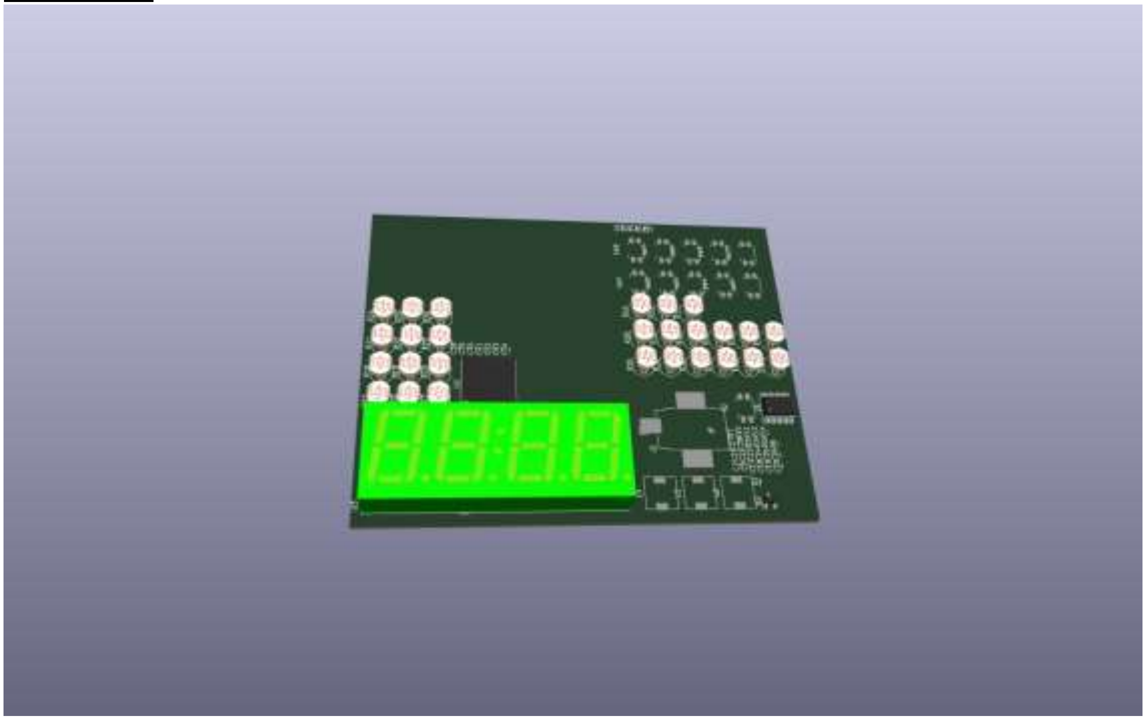




PCB LAYOUTDESIGN 2(AJAY)



<u>3D VIEWER</u>



8 Sources/References

https://github.com/DOLUWAMU-TAIWO/TD/tree/main/HARDWARE_Engineering