Smart Combination Lock

Digital System Design Final Project

Yiwen Ying
Dept. of Electronic and Electrical Engineering
Southern University of Science and Technology
ShenZhen, China
12210159@mail.sustech.edu.cn

Jingjun Xu
Dept. of Electronic and Electrical Engineering
Southern University of Science and Technology
ShenZhen, China
12210357@mail.sustech.edu.cn

***Abstract*—A smart combination lock system has been designed and implemented to enhance security through encryption-supported password management and user feedback mechanisms. The system architecture supports functionalities such as password input, password verification, encrypted storage, anti-brute force protection, and alarm triggering via both sound and Bluetooth. The methodology includes detailed hardware design, VHDL module block diagrams, and an ASM Chart that defines the state transitions between $lock$, $unlock$, set_pwd, and $check\_pwd$ states. Key modules, including segment display, timer, button debounce, buzzer, Bluetooth, encrypt, and decrypt, are discussed with critical code segments highlighted to demonstrate modularity and system integration. Simulation and real-world testing, performed through some dedicated testbench, validate the system's functionality and robustness. Results indicate that the smart combination lock operates reliably, with accurate password verification and effective alarm triggering in response to unauthorized attempts. This work demonstrates the feasibility of implementing a secure and interactive locking mechanism using a modular, hardware-based approach.**

***Index Terms*—Smart combination lock, VHDL, State machine**

I. Introduction

Combination locks are widely used in several fields due to their convenience and security. They are commonly found in family homes, offices, private facilities, and more, and are used to increase the security of devices. Smart combination locks are particularly favored, as they combine traditional combination locks with modern smart technologies, such as remote control, to provide a higher level of security.

The Smart Digital Combination Lock System, based on the simple digital combination lock, combines a variety of security features and is committed to providing users with a safer yet convenient password protection solution. The system will ensure that it provides timely feedback and effective security at key points such as password entry, incorrect attempts, and system lockout.

The system has those functions:

- **Efficient password input:** The system is able to clearly display the input numbers in real time through the switch.
- **Secure password storage:** The system uses encryption when storing passwords to improve security.
- **Intelligent Error Attempts:** The system will count the number of error attempts, and if the error is more than four times, the system will automatically lock for two minutes to prevent violent cracking.
- **Clear right and wrong feedback:** When the system verifies the password, it informs the user of the right and wrong passwords through the different states of digital tubes, LED lights and buzzers.
- **Remote monitoring and alert:** The system will support remote monitoring and alert function, when the number of incorrect attempts reaches four, the system will send the alert message to the user's cell phone application via Bluetooth module.

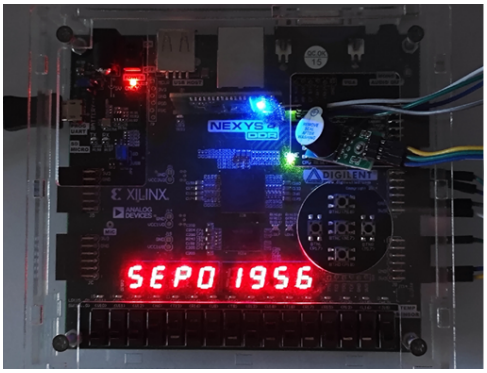


Fig. 1: System Overview

II. Methodology

A. System Overview

In case of hardware, we use Nexys4 DDR, bluetooth module, and buzzer module.

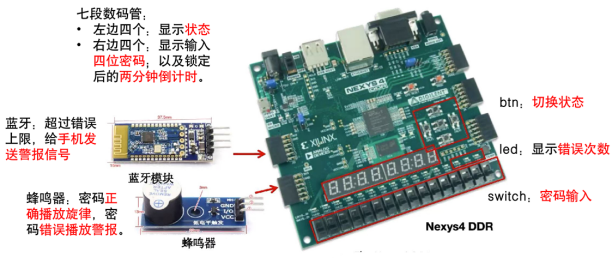


Fig. 2: Hardware Connection

The codes are organized by top module, with each function realized in each module.

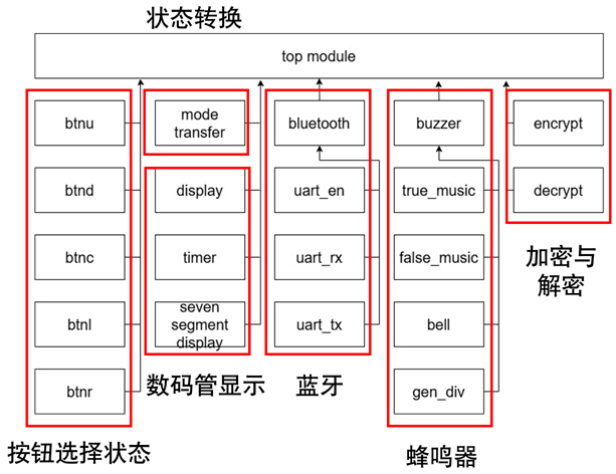


Fig. 3: Code Organization

The RT Level Schematic is as follows:

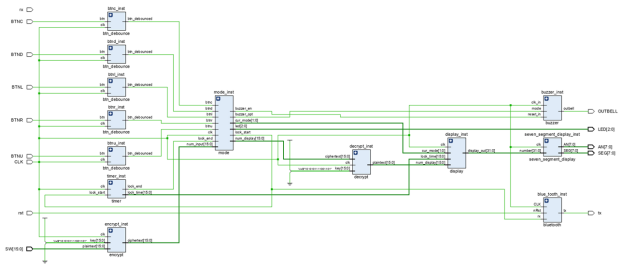


Fig. 4: RTL Schematic

B. VHDL Module Block Diagram

The block diagram of the VHDL module is illustrated in the figure below:

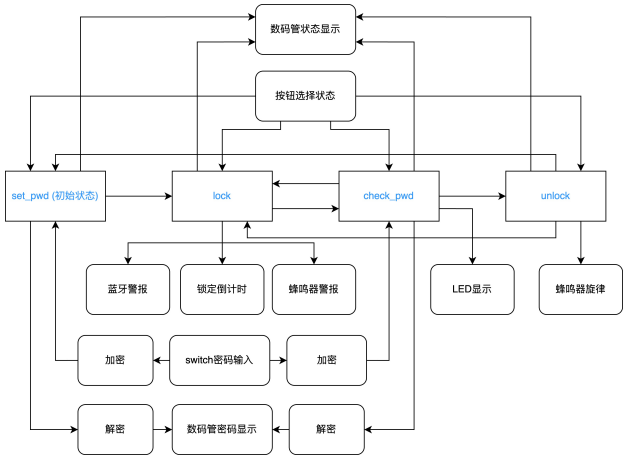


Fig. 5: VHDL Module Block Diagram

As shown in the figure, there are four states in our system: set_pwd, $lock$, $check\_pwd$, and $unlock$.

- The initial state is the set_pwd. After setting a password, the user can transition to the $lock$ state to lock the system.
- In the $lock$ state, the user can transition to the $check\_pwd$ state to enter a password and attempt to unlock the system.
- In the $check\_pwd$ state, the user has three chances to enter the password incorrectly. If the user successfully enters the correct password, the system will transition to the $unlock$ state. However, if the user enters the wrong password four times in a row, the system will transition back to the $lock$ state and the system will be locked for two minutes.
- In the $unlock$ state, the user can transition to the $lock$ state or the set_pwd state.

Various functional modules have been implemented within the four states to support the system's operation.

- **Button**: It facilitates the transitions between the four states.
- **Segment Display**:
  - For the four segment display positions on the left: It displays the four states of the system.
  - For the four segment display positions on the right: It shows the current input password when the system is in the set_pwd state and the $check\_state$ state. Also, it shows the countdown time when the system is locked for two minutes in the $lock$ state.
- **Switch**: It facilitates the password input from the user in set_pwd state and the $check\_pwd$ state.
- **Encrypt**: It encrypts the user-entered password and stores it in the system as ciphertext in set_pwd state and the $check\_pwd$ state.
- **Decrypt**: It decrypts the stored ciphertext to reveal the user-entered password in set_pwd state and the $check\_pwd$ state.
- **Timer**: It countdowns the time when the system is locked for two minutes in $lock$ state.
- **Bluetooth**: It sends an alert message to the owner of the combination lock to notify them that there may have been an unauthorized attempt to unlock the system, as the password was entered incorrectly four times in a row in the $check\_pwd$ state.
- **Buzzer**: In the $check\_pwd$ state, it plays an alarm when the password is entered incorrectly four times in a row, alerting the owner of the combination lock that there may have been an unauthorized attempt to unlock the system. Furthermore, it plays a sound to indicate that the correct password has been entered.
- **LED**: It informs the user of the total number of incorrect password attempts while in the $check\_pwd$ state.

C. ASM Chart

The ASM chart illustrates the details of the transitions between the four states.

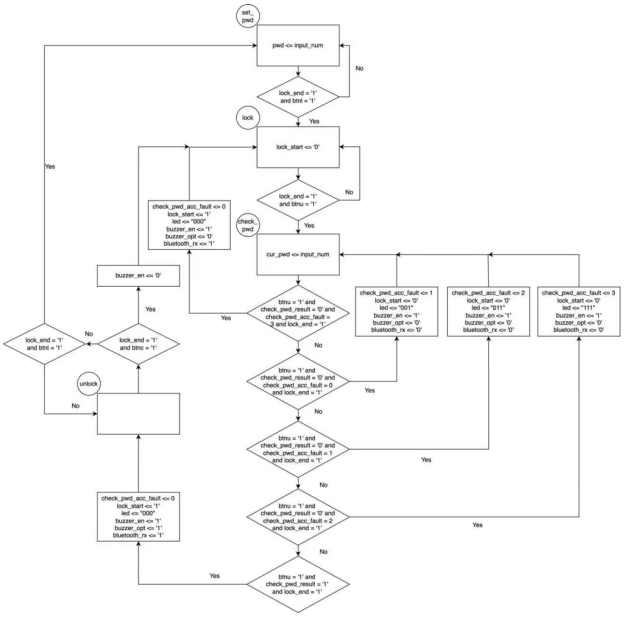


Fig. 6: ASM Chart

The state transfer logic is as follows:

```
case mode_reg is
    when 0 =>
        if btnu = '1' and lock_end = '1' then -- lock -> check_pwd
            check_pwd_acc_fault <= 0;
            mode_reg <= 3;
        end if;
        lock_start_reg <= '0';
        led_reg <= "000";

    when 1 =>
        if btnc = '1' and lock_end = '1' then
            mode_reg <= 0;
            buzzer_en <= '0';
        elsif btnl = '1' and lock_end = '1' then
            mode_reg <= 2;
            buzzer_en <= '0';
        end if;
        led_reg <= "000";

    when 2 =>
        if btnl = '1' and lock_end = '1' then
            mode_reg <= 0;
        end if;
        led_reg <= "000";

    when 3 =>
        if btnu = '1' and lock_end = '1' and check_pwd_result = '0' and check_pwd_acc_fault = 3 then -- fail last time and accumulate three times
            check_pwd_acc_fault <= 0;
            mode_reg <= 0;
            lock_start_reg <= '1';
            led_reg <= "000";
            buzzer_en <= '1';
            buzzer_opt <= '0';
        elsif btnu = '1'and lock_end = '1' and check_pwd_result = '0' and check_pwd_acc_fault = 0 then
            check_pwd_acc_fault <= 1;
            mode_reg <= 3;
            lock_start_reg <= '0';
            led_reg <= "001";
            buzzer_en <= '1';
            buzzer_opt <= '0';
        elsif btnu = '1'and lock_end = '1' and check_pwd_result = '0' and check_pwd_acc_fault = 1 then
            check_pwd_acc_fault <= 2;
            mode_reg <= 3;
            lock_start_reg <= '0';
            led_reg <= "011";
            buzzer_en <= '1';
            buzzer_opt <= '0';
        elsif btnu = '1'and lock_end = '1' and check_pwd_result = '0' and check_pwd_acc_fault = 2 then
            check_pwd_acc_fault <= 3;
            mode_reg <= 3;
            lock_start_reg <= '0';
            led_reg <= "111";
            buzzer_en <= '1';
            buzzer_opt <= '0';
        elsif btnu = '1' and lock_end = '1' and check_pwd_result = '1' then
            mode_reg <= 1;
            lock_start_reg <= '0';
            led_reg <= "000";
            buzzer_en <= '1';
            buzzer_opt <= '1';
        end if;
end case;
```

D. Segement Display

The seven segment display is divided into two sections: the left four digits are used to show different states (such as ”LC”, ”OK”, ”SEPD”, ”CHPD”), while the right four digits are used to display numerical values. These values can represent the countdown time during a timing operation, or digits of a password when setting or entering a password.

```
lock_time_display <= (others => '1') when lock_time = x"0000" else lock_time;

with cur_mode select
    display_out <=
        L & C & x"FF" & lock_time_display when 0,
        "0000" & H & x"FFFFFF" when 1,
        "0101" & E & P & "0000" & num_display when 2,
        C & H & P & "0000" & num_display when 3,
        x"FFFFFFFF" when others;
```

The seven-segment display operates on the principle of multiplexing, which allows multiple digits to be displayed using a single set of segment drivers by rapidly switching between them. The display consists of eight digits, each composed of seven segments (plus an optional decimal point), which can be individually controlled to form various characters and numbers.

In VHDL implementation, a counter generates timing signals to select which digit is currently being displayed. The counter increments on each clock cycle, and its higher-order bits are used to select the active digit (anode selection). A decoder converts the counter’s output into a signal that activates one of the eight digits at a time, while the other digits are turned off. Simultaneously, the corresponding digit data is selected from the input number and converted into the appropriate segment signals to display the correct character

on the active digit. This process repeats rapidly, creating the illusion that all digits are displayed simultaneously, even though only one digit is actually active at any given moment.

```
-- 3-8 decoder
with anode_select select
    an_tmp <=
        "11111110" when "000",
        "11111101" when "001",
        "11111011" when "010",
        "11110111" when "011",
        "11101111" when "100",
        "11011111" when "101",
        "10111111" when "110",
        "01111111" when "111",
        "11111111" when others;
```

```
-- the number to be displayed
with anode_select select
    digit_data <=
        number(3 downto 0) when "000",
        number(7 downto 4) when "001",
        number(11 downto 8) when "010",
        number(15 downto 12) when "011",
        number(19 downto 16) when "100",
        number(23 downto 20) when "101",
        number(27 downto 24) when "110",
        number(31 downto 28) when "111",
        "0000" when others;
```

```
-- gfedcba
with digit_data select
    segment_data <=
        "1000000" when "0000", -- 0
        "1111001" when "0001", -- 1
        "0100100" when "0010", -- 2
        "0110000" when "0011", -- 3
        "0011001" when "0100", -- 4
        "0010010" when "0101", -- 5
        "0000010" when "0110", -- 6
        "1111000" when "0111", -- 7
        "0000000" when "1000", -- 8
        "0010000" when "1001", -- 9
        "1000110" when "1010", -- C
        "0000110" when "1011", -- E
        "1000111" when "1100", -- L
        "0001001" when "1101", -- H
        "0001100" when "1110", -- P
        "1111111" when others;
```

E. Timer

The design of the timer module is based on the principle of digital logic and timing control, and is driven by clock signals to achieve accurate time counting and countdown functions. The core principle is to use the combination of clock division and counter to convert the high-frequency clock signal into a time unit of seconds, and on this basis, the time is decremented counting.

When `lock_start` is set to a high level, the timer is activated, and the internal clock divider and counters begin to operate. The clock divider functions to reduce the frequency of the high-frequency input clock signal (typically at the MHz level) by counting, thereby generating a second-level time base signal. The second-level counter is responsible for recording the current number of seconds and works in conjunction with the minute counter. Each time the clock divider triggers a second-level update, the second counter is decremented by 1. This process is repeated until both the minute and second counters are reduced to zero, at which point the timing ends and the output signal `lock_end` is set to a high level, indicating the completion of the timing.

To facilitate intuitive display and processing of the timing duration, the timer further decomposes the minutes and seconds into tens and units digits, which are then mapped to different segments of the 16-bit output signal `lock_time`. This allows the current timing state to be clearly represented in binary form and enables the timing information to be directly read and displayed by a seven-segment display.

```
process(clk)
begin
    if rising_edge(clk) then
        if lock_start = '1' then
            clock_divider <= 0;
            seconds_reg <= 0;
            minutes_reg <= 2;
            timer_running <= true;
            lock_end_reg <= '0';
        elsif timer_running then
            if clock_divider = 99999999 then
                clock_divider <= 0;
                if seconds_reg = 0 then
                    if minutes_reg = 0 then
                        timer_running <= false;
                        lock_end_reg <= '1';
                    else
                        minutes_reg <= minutes_reg
                        ↪  - 1;
                        seconds_reg <= 59;
                    end if;
                else
                    seconds_reg <= seconds_reg - 1;
                end if;
            else
                clock_divider <= clock_divider + 1;
            end if;
        else
            clock_divider <= 0;
        end if;
    end if;
end process;
```

```
sec_tens <= seconds_reg / 10;
sec_units <= seconds_reg mod 10;
min_tens <= minutes_reg / 10;
min_units <= minutes_reg mod 10;

lock_time(15 downto 12) <=
↪  std_logic_vector(to_unsigned(min_tens, 4));
lock_time(11 downto 8) <=
↪  std_logic_vector(to_unsigned(min_units, 4));
lock_time(7 downto 4) <=
↪  std_logic_vector(to_unsigned(sec_tens, 4));
lock_time(3 downto 0) <=
↪  std_logic_vector(to_unsigned(sec_units, 4));

lock_end <= lock_end_reg;
```

F. Button Debounce

In actual hardware circuits, mechanical buttons generate jitter signals when pressed or released. These jitter signals can lead to false triggering, e.g. in counters or state machines that can be incorrectly identified as multiple presses. Therefore, it is necessary to filter out these jitter signals through debounce logic to ensure the stability and reliability of the button signal.

- **Signal Synchronization:** A synchronization register `btn_sync` is used to synchronize the button signal to prevent metastability issues. `btn_sync` is a 2-bit register that shifts the button signal on each rising edge of the clock to ensure signal stability.
- **Debounce Logic:** A counter `counter` is used to detect the stability of the button signal. If the button signal remains unchanged for a certain period of time (determined by the counter's upper limit), the signal is considered stable and is assigned to `btn_stable`.
- **Button Output Processing:** The rising edge of the `btn_stable` signal is detected. `btn_debounced` is set to 1 only when `btn_stable` transitions from 0 to 1; otherwise, it is set to 0.

```
process(clk)
begin
    if rising_edge(clk) then
        -- Update btn_sync in this process only
        btn_sync <= btn_sync(0) & btn;

        -- Debounce logic
        if btn_sync(1) /= btn_stable then
            if counter = 2_000_000 then
                btn_stable <= btn_sync(1);
                counter <= (others => '0');
            else
                counter <= counter + 1;
            end if;
        else
            counter <= (others => '0');
        end if;
    end if;
end process;
```

G. Buzzer

A buzzer operates by converting electrical energy into sound energy using a piezoelectric or electromagnetic mechanism. When an alternating voltage is applied, the piezoelectric element or coil vibrates, generating an audible tone.

In our system, the buzzer module is supported by four submodules, each responsible for a specific function to ensure accurate and reliable sound generation:

- *music_correct*: This submodule handles the playback of musical notes when the correct password is entered. It sequences the notes according to a predefined rhythm and sends them to the bell module at one-second intervals, enabling a clear and pleasant confirmation sound.
- *music_wrong*: Similar to *music_correct*, this submodule manages the playback of musical notes when the entered password is incorrect. It generates the notes according to a warning rhythm and sends them to the bell module at one-second intervals to indicate a failed attempt.
- *gen_div*: This submodule is responsible for dividing the 100 MHz system clock down to a lower frequency clock signal suitable for driving the bell module. It ensures that the bell module receives a stable and consistent clock input for precise note playback.
- *bell*: This submodule receives the note signals and the clock input, and generates output signals at specific frequencies that correspond to the musical notes. It achieves this by using either a frequency divider or a waveform generator, producing sound at the appropriate pitch for each note.

Buzzer

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

entity buzzer is
    port
    (
        clk_in:in std_logic;
        reset_in :in std_logic;
        mode:in std_logic; -- 0 wrong1right
        outbell:out std_logic
    );
end entity;

architecture lin of buzzer is
signal bells:std_logic;
signal clk_tmp:std_logic;
signal clk_tmp2:std_logic;
signal key_in:std_logic_vector(2 downto 0);
signal key_wrong: std_logic_vector(2 downto 0);
signal key_right: std_logic_vector(2 downto 0);
signal pre_div:std_logic_vector(15 downto 0);
signal not_reset_in:std_logic;

component false_music is
    port
    (
        clk:in std_logic;--
        rst: in std_logic;
        key_output:out std_logic_vector(2 downto 0)
    );
end component;

component true_music is
    port
    (
        clk:in std_logic;--
        rst: in std_logic;
        key_output:out std_logic_vector(2 downto 0)
    );
end component;

component gen_div is
generic(div_param:integer:=2);
port
(
    clk:in std_logic;
    bclk:out std_logic;
    resetb:in std_logic
);
end component;

component bell is
port
(
    clkin:in std_logic;
    resetin :in std_logic;
    key:in std_logic_vector(2 downto 0);
    bell_out:out std_logic
);
end component;

begin
outbell <=bells when reset_in = '1' else '1';
key_in <= key_wrong when mode = '0' else key_right;
not_reset_in <= not reset_in;
```

```
music_correct:
        true_music port map
        (
        clk=>clk_in,
        rst=>reset_in,
        key_output=>key_right
        );

music_wrong:
        false_music port map
        (
        clk=>clk_in,
        rst=>reset_in,
        key_output=>key_wrong
        );

gen_10M:
        gen_div generic map(2)
        port map
        (
            clk=>clk_in,
            resetb => not_reset_in,
            bclk=>clk_tmp
        );

bell8s:
        bell port map
        (
        clkin=>clk_tmp,
        resetin =>reset_in,
        key=>key_in,
        bell_out=>bells
        );

end lin;
```

music_correct

```
entity true_music is
    port
    (
        clk:in std_logic;
        rst: in std_logic;
        key_output:out std_logic_vector(2 downto 0)
    );
end entity;

architecture Behavioral of true_music is
type integer_array is array (natural range <>) of
↪  integer;
constant len : integer := 4;
constant music: integer_array(0 to len-1) := (1, 3,
↪  5, 7);
signal note_cnt: unsigned(1 downto 0);

constant max_count : integer := 99_999_999;
signal counter : unsigned(26 downto 0) := (others
↪  => '0');

signal key_temp : std_logic_vector(2 downto 0) :=
↪  "000";
begin
    process(clk, rst)
    begin
        if (rst = '0') then
            key_temp <= "000";
        elsif (rising_edge(clk)) then
            if(counter = max_count) then
                counter <= (others => '0');
                if (note_cnt mod len = 0) then
                    key_temp <= std_logic_vector
                    (TO_UNSIGNED(music(0), 3));
                elsif (note_cnt mod len = 1) then
                    key_temp <= std_logic_vector
                    (TO_UNSIGNED(music(1), 3));
                elsif (note_cnt mod len = 2) then
                    key_temp <= std_logic_vector
                    (TO_UNSIGNED(music(2), 3));
                elsif (note_cnt mod len = 3) then
                    key_temp <= std_logic_vector
                    (TO_UNSIGNED(music(3), 3));
                end if;
                note_cnt <= note_cnt + 1;
            else
                counter <= counter + 1;
            end if;
        end if;
    end process;

    key_output <= key_temp;
end Behavioral;
```

music_wrong

```
entity false_music is
    port
    (
        clk:in std_logic;
        rst: in std_logic;
        key_output:out std_logic_vector(2 downto 0)
    );
end entity;

architecture Behavioral of false_music is
begin
    key_output <= "111";
end Behavioral;
```

gen_div

```
entity  gen_div is
    generic(div_param:integer:=1);
    port
    (
        clk:in std_logic;
        bclk:out std_logic;
        resetb:in std_logic
    );
end gen_div;

architecture behave of gen_div is
signal tmp:std_logic;
signal cnt:integer range 0 to div_param:=0;

begin
    process(clk,resetb)
    begin
        if resetb='1' then
            cnt<=0;
            tmp<='0';
        elsif rising_edge(clk) then
            cnt<=cnt+1;
            if cnt=div_param-1 then
                tmp<=not tmp;
                cnt<=0;
            end if;
        end if;
    end process;
    bclk<=tmp;
end behave;
```

bell

```
entity bell is
    port
    (
        clkin:in std_logic;
        resetin :in std_logic;
        key:in std_logic_vector(2 downto 0);
        bell_out:out std_logic
```

```
    );
end entity;

architecture behave of bell is
signal bell_tmp:std_logic;
signal pre_div:std_logic_vector(15 downto 0);
begin
bell_out<=bell_tmp;
    process(clkin,key,resetin)
    variable cnt:std_logic_vector(15 downto 0):=X"0000";
    begin
        if resetin='0' then
            bell_tmp<='0';
            cnt:=X"0000";
            pre_div<=X"0000";--0
        else
            if rising_edge(clkin) then
                if cnt>=pre_div then
                    bell_tmp<=not bell_tmp;
                    cnt:=X"0000";
                    if key="000" then
                        pre_div<=X"0000";--0
                    elsif key="001" then
                        pre_div<=X"4AA7";--1
                    elsif key="010" then
                        pre_div<=X"4282";--2
                    elsif key="011"then
                        pre_div<=X"3B41";--3
                    elsif key="100" then
                        pre_div<=X"37ED";--4
                    elsif key="101"then
                        pre_div<=X"31D3";--5
                    elsif key="110" then
                        pre_div<=X"2C64";--6
                    elsif key="111" then
                        pre_div<=X"278C";--7
                    end if;
                else
                    cnt:=cnt+'1';
                end if;
            end if;
        end if;
    end process;
end behave;
```

```
    );
    Port(
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        rx_en : out STD_LOGIC;
        tx_en : out STD_LOGIC
    );
end uart_en;

architecture Behavioral of uart_en is
    signal cnt_tx : integer range 0 to sysclk/BPS-1 := 0;
    constant bps_tx : integer := sysclk/BPS;
    signal cnt_rx : integer range 0 to sysclk/(BPS*8)-1 := 0;
    constant bps_rx : integer := sysclk/(BPS*8);
begin
    process(clk, rst)
    begin
        if rst = '0' then
            cnt_tx <= 0;
        elsif rising_edge(clk) then
            if cnt_tx >= bps_tx - 1 then
                cnt_tx <= 0;
            else
                cnt_tx <= cnt_tx + 1;
            end if;
        end if;
    end process;
    tx_en <= '1' when cnt_tx = bps_tx - 1 else '0';

    process(clk, rst)
    begin
        if rst = '0' then
            cnt_rx <= 0;
        elsif rising_edge(clk) then
            if cnt_rx >= bps_rx - 1 then
                cnt_rx <= 0;
            else
                cnt_rx <= cnt_rx + 1;
            end if;
        end if;
    end process;
    rx_en <= '1' when cnt_rx = bps_rx - 1 else '0';
end Behavioral;
```

H. Bluetooth UART

The Bluetooth module in our system is implemented using a UART (Universal Asynchronous Receiver-Transmitter) interface, which enables wireless communication between devices. The UART interface consists of three main components: `uart_en`, `uart_rx`, and `uart_tx`, each serving a specific purpose.

The `uart_en` module generates the enable signals for the UART transmitter and receiver. It takes the system clock (`clk`) and reset signal (`rst`) as inputs and produces `rx_en` and `tx_en` signals. The baud rate (`BPS`) and system clock frequency (`sysclk`) are configurable. The module uses counters to generate the enable signals at the specified baud rate, ensuring proper timing for data transmission and reception.

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

entity uart_en is
    generic(
        BPS : integer := 115200;
        sysclk : integer := 100_000_000
```

UART communication follows a standard data format:

- **Start Bit**: A single low (0) bit indicating the start of a data frame.
- **Data Bits**: Typically 8 bits of data, transmitted least significant bit first.
- **Parity Bit** (optional): A single bit for error detection, configurable as None, Odd, or Even.
- **Stop Bit(s)**: One or two high bits marking the end of a data frame.

The `uart_rx` and `uart_tx` modules operate based on the UART data format. The `uart_rx` module detects the start bit, samples the data bits, checks the parity bit (if enabled), and verifies the stop bit(s). The `uart_tx` module generates the start bit, transmits the data bits, adds the parity bit (if enabled), and appends the stop bit(s).

Both the rx and tx module are realized by finite state machine.

```
case state is
    when "0000" => -- idle
        rx_data_r <= (others => '0');
        rx_data_vld_r <= '0';
        sample_cnt <= 0;
```

```
        if rx_edge = '1' then
            state <= "0001";
        end if;
    when "0001" => -- start
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 3 then
            sample_cnt <= 0;
            state <= "0010";
        end if;
    when "0010" => -- s1
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            rx_data_r(0) <= rx_r(1);
            state <= "0011";
        end if;
    when "0011" => -- s2
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            rx_data_r(1) <= rx_r(1);
            state <= "0100";
        end if;
    when "0100" => -- s3
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            rx_data_r(2) <= rx_r(1);
            state <= "0101";
        end if;
    when "0101" => -- s4
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            rx_data_r(3) <= rx_r(1);
            state <= "0110";
        end if;
    when "0110" => -- s5
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            rx_data_r(4) <= rx_r(1);
            state <= "0111";
        end if;
    when "0111" => -- s6
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            rx_data_r(5) <= rx_r(1);
            state <= "1000";
        end if;
    when "1000" => -- s7
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            rx_data_r(6) <= rx_r(1);
            state <= "1001";
        end if;
    when "1001" => -- s8
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            rx_data_r(7) <= rx_r(1);
            if PARITY = "NONE" then
                state <= "1100";
            elsif PARITY = "ODD" then
                state <= "1010";
            else
                state <= "1011";
            end if;
        end if;
    when "1010" => -- parity_odd
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 and rx_r(1) = not odd
        ↪ then
            sample_cnt <= 0;
            state <= "1100";
        end if;
    when "1011" => -- parity_even
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 and rx_r(1) = odd then
            sample_cnt <= 0;
            state <= "1100";
        end if;
    when "1100" => -- stop1
        if STOP = 1 then
            sample_cnt <= sample_cnt + 1;
            if sample_cnt >= 7 then
                sample_cnt <= 0;
                state <= "0000";
                rx_data <= rx_data_r;
                rx_data_vld_r <= '1';
            end if;
        else
            sample_cnt <= sample_cnt + 1;
            if sample_cnt >= 7 and rx_r(1) = '1'
            ↪ then
                state <= "1101";
                sample_cnt <= 0;
            end if;
        end if;
    when "1101" => -- stop2
        sample_cnt <= sample_cnt + 1;
        if sample_cnt >= 7 then
            sample_cnt <= 0;
            state <= "0000";
            rx_data <= rx_data_r;
            rx_data_vld_r <= '1';
        end if;
    when others => null;
end case;
```

The transmitting part of UART is similar with receiving part, so it's omittied in the report.

I. Encrypt and Decrypt

The Feistel cipher is a symmetric block cipher that uses a series of rounds to transform plaintext into ciphertext. The same structure is used for both encryption and decryption. This report summarizes the mathematical principles of the Feistel cipher.

The **encryption** process of the Feistel cipher involves the following steps:

- Given a 16-bit key K, four 8-bit subkeys K_1, K_2, K_3, K_4 are generated:

$$K_i = \text{left\_rotate}(K, 4i)[15:8] \oplus i \tag{1}$$

- The plaintext P is divided into two 8-bit halves L_0 and R_0. For 4 rounds, the following operations are performed:

$$L_i = R_{i-1}, \quad R_i = L_{i-1} \oplus F(R_{i-1}, K_i) \tag{2}$$

  where $F(R, K) = \text{rotate\_left}((R \oplus K) + 5, 1)$.
- The ciphertext C is formed by concatenating the final halves:

$$C = R_4 \parallel L_4 \tag{3}$$

The **decryption** process of the Feistel cipher is similar to the encryption process but operates in reverse order. The steps are as follows:

- Starting from the ciphertext C:

$$R_4 = C_{15:8}, \quad L_4 = C_{7:0} \tag{4}$$

- For 4 rounds, the following operations are performed in reverse order:

$$R_{i-1} = L_i, \quad L_{i-1} = R_i \oplus F(L_i, K_i) \tag{5}$$

- The plaintext P is recovered by:

$$P = L_0 \parallel R_0 \tag{6}$$

The main functions are shown below:

For each encryption and decryption loop, we use a subkey to do the transformation, for more safety.

```
function GenerateSubkeys(input_key :
↪  STD_LOGIC_VECTOR(15 downto 0)) return
↪  subkey_array is
    variable keys     : subkey_array;
    variable temp_key : STD_LOGIC_VECTOR(15 downto
    ↪  0) := input_key;
begin
    for i in 1 to NUM_ROUNDS loop
        temp_key := temp_key(11 downto 0) &
        ↪  temp_key(15 downto 12);
        keys(i) := temp_key(15 downto 8) xor
        ↪  std_logic_vector(to_unsigned(i, 8));
    end loop;
    return keys;
end function;
```

For safety concern, we use nonlinear transformation in encryption and decryption. In each loop, xor operation is performed between the subkey and half of the password.

```
function F(right : STD_LOGIC_VECTOR(7 downto 0);
↪  key : STD_LOGIC_VECTOR(7 downto 0)) return
↪  STD_LOGIC_VECTOR is
    variable temp : unsigned(7 downto 0);
    variable rotated : unsigned(7 downto 0);
begin
    temp := unsigned(right) xor unsigned(key);
    temp := temp + 5;
    rotated := temp(6 downto 0) & temp(7);
    return std_logic_vector(rotated);
end function;
```

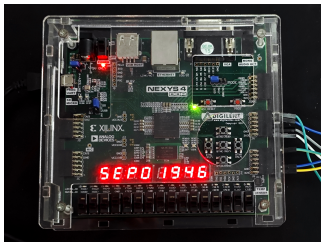
With the functions above, the encryption and decryption operation can be easily done.

```
subkeys <= GenerateSubkeys(key);

process(clk)
    variable i : integer;
    variable temp_L, temp_R : STD_LOGIC_VECTOR(7
    ↪  downto 0);
begin
    if rising_edge(clk) then
        left_round(0) <= plaintext(15 downto 8);
        right_round(0) <= plaintext(7 downto 0);
        for i in 1 to NUM_ROUNDS loop
            temp_L := right_round(i-1);
            temp_R := left_round(i-1) xor
            ↪  F(right_round(i-1), subkeys(i));
            left_round(i) <= temp_L;
            right_round(i) <= temp_R;
        end loop;
    end if;
end process;

ciphertext <= right_round(NUM_ROUNDS) &
↪  left_round(NUM_ROUNDS);
```

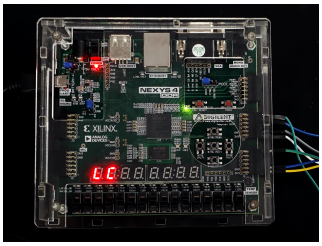
III. Result and Analysis

A. States

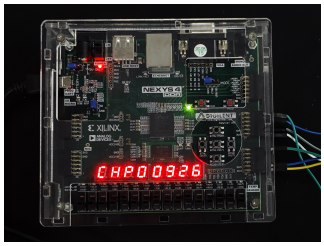
For the four states (set password, check password, lock, unlock), we have different seven segement display effect, led display, buzzer result, and bluetooth result.



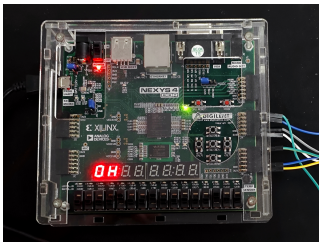
(a) Set Password



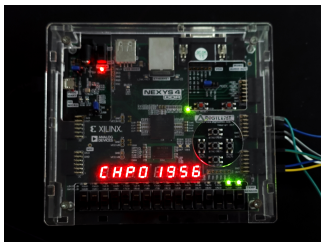
(b) Lock



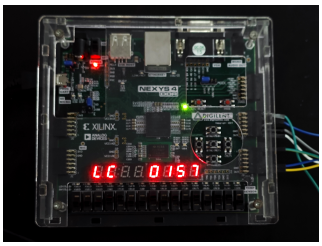
(c) Check Password



(d) Correct, Unlock



(e) Mistake 2 Times



(f) Mistake Out of Limit

Fig. 7: Process of Password Operations

B. Encrypt and Decrypt

Testing the encryption and decryption module by testbench, we can verify that both the modules are right, and text after decryption is the same as the plain text.

We test three sets of data:

```
constant test_vectors : test_array_type := (
    (plaintext => x"1234", key => x"ABCD"),
    (plaintext => x"5678", key => x"1111"),
    (plaintext => x"9ABC", key => x"FFFF")
);
```

The testbench can automatically test all the cases:

```
test_process : process
begin
    for i in test_vectors'range loop
        plaintext <= test_vectors(i).plaintext;
        key <= test_vectors(i).key;
        wait for clk_period * 10;
        if decrypted = test_vectors(i).plaintext
        ↪  then
            report "  Result      : PASS" severity
            ↪  note;
```

```
        else
            report "  Result      : FAIL" severity
            ↪  error;
        end if;
        wait for clk_period * 2;
    end loop;
    report "All tests finished.";
    wait;
end process;
```

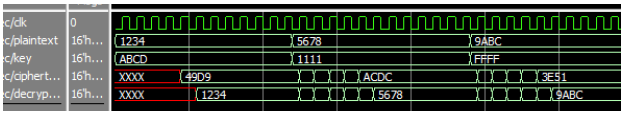


Fig. 8: Simulation Result of Encryption and Decryption

It works very quickly, which shows the hardaware acceleration capabilities of FPGA.

C. Bluetooth

In simulation, we test many cases:

```
constant test_data : test_data_array := (
x"AA", x"1F", x"55", x"88", x"FF", x"37", x"7E",
↪  x"2B", x"4D", x"9A"
);
```

To validate the function, we connect rx and tx, and run uart_en, uart_rx, uart_tx together.

```
-- UART receiver module instantiation
uart_rx_inst : uart_rx
    generic map(
        PARITY => "NONE",
        STOP => 1
    )
    port map(
        clk => clk,
        rst => rst,
        rx_en => rx_en,
        tx_en => tx_en,
        rx => loopsignal, -- Connect RX to loopback
        ↪  signal
        rx_data => rx_data,
        rx_end => rx_end,
        rx_data_vld => rx_data_vld
    );

-- UART transmitter module instantiation
uart_tx_inst : uart_tx
    generic map(
        PARITY => "NONE",
        STOP => 1
    )
    port map(
        clk => clk,
        rst => rst,
        tx_data_vld => tx_data_vld,
        tx_data => tx_data,
        tx_en => tx_en,
        tx_ack => tx_ack,
        tx => loopsignal -- Connect TX to loopback
        ↪  signal
    );
-- Test process
test_process : process
begin
    -- Initialize
    rst <= '1'; -- Start the system
    wait for CLK_PERIOD;
    rst <= '0'; -- Hold reset
    wait for CLK_PERIOD;
    rst <= '1'; -- Release reset
    wait for CLK_PERIOD;

    -- Test UART transmit and receive (loopback)
    report "Starting UART loopback test";

    -- Loop through test data array
    for i in test_data'range loop
        -- Prepare data to send
        tx_data <= test_data(i);
        tx_data_vld <= '1';
        wait until tx_ack = '1'; -- Wait for
        ↪  transmitter to acknowledge
        tx_data_vld <= '0';
        -- Wait for the received data to be valid
        wait until rx_data_vld = '1';
        wait for BIT_PERIOD; -- Wait for the next
        ↪  bit
    end loop;

    -- End simulation
    report "Simulation completed successfully";
    wait;
end process;
```

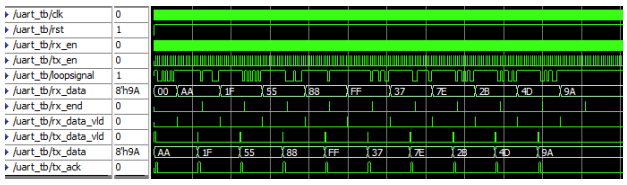


Fig. 9: Simulation of Bluetooth

The module can successfully transmit and receive data.

In real-world experiments, the module can transmit signals by UART. On the application side on the phone, it can receive the message ”00”.

Fig. 10: Bluetooth Message Received on the Phone

Fig. 11: Bluetooth Module Operate Normally

D. Buzzer

The purpose of the buzzer module is to play different sounds in different states. Here, two main states are considered: an

alarm is played when the password is entered incorrectly, and music is played when the password is entered correctly.

We tested the above two situations respectively. Here, the variable $mode$ represents 0 when the alarm is playing and 1 when the music is playing. To more clearly reflect the pitch of the played notes, here we directly output the pitch itself instead of output the changing signal with a certain frequency.

The testbench is shown as follows.

```
architecture tb of top_tb is
    signal clk_in: std_logic := '0';
    signal reset_in: std_logic := '1';
    signal mode: std_logic := '0';
    signal key_in:std_logic_vector(2 downto 0);
    constant clk_period: time := 10 ns;

    component buzzer is
        port
        (
            clk_in:in std_logic;
            reset_in :in std_logic;
            mode:in std_logic;
            keyin: out std_logic_vector(2 downto 0)
        );
    end component;

begin
    uut: buzzer
        port map (
            clk_in =&gt; clk_in,
            reset_in =&gt; reset_in,
            mode =&gt; mode,
            keyin =&gt; key_in
        );
    clk_gen: process
    begin
        clk_in &lt;= '0';
        wait for clk_period/2;
        clk_in &lt;= '1';
        wait for clk_period/2;
    end process clk_gen;

    stimulus: process
    begin
        reset_in &lt;= '1';
        wait for 10000 ns;
        reset_in &lt;= '0';
        wait for 10000 ns;
        reset_in &lt;= '1';

        mode &lt;= '0';
        wait for 100000 ns;
        mode &lt;= '1';
        wait for 100000 ns;
        mode &lt;= '0';
        wait for 100000 ns;

        wait;
    end process stimulus;
end tb;
```

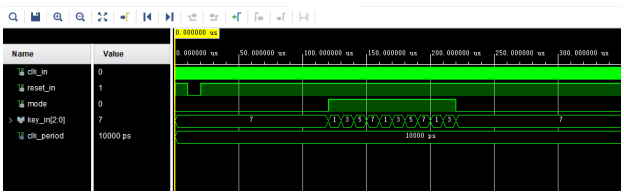


Fig. 12: Simulation of Buzzer

According to the simulation results, our buzzer module responds correctly in both scenarios. When the $mode$ is set to 0, the output node is 7, which represents the alert signal. When the $mode$ is 1, the output node outputs the sequence $1 - 3 - 5 - 7$ iteratively, representing our custom-designed melody.

IV. Conclusion

In this project, the development of a smart combination lock system has been successfully accomplished, integrating multiple advanced functionalities to enhance security and user experience. The system supports comprehensive features including:

- **Password Management**: Users can set and modify passwords with ease, ensuring customizable security.
- **Encrypted Storage**: Passwords are stored in an encrypted format to protect user data against unauthorized access.
- **Password Verification**: The system verifies user-entered passwords, allowing access only when the correct password is provided.
- **Anti-Brute Force Protection**: After a predefined number of failed attempts, the system triggers security measures to prevent brute force attacks.
- **Bluetooth Remote Alarm**: The system is equipped with Bluetooth connectivity, enabling the owner to receive remote alarm notifications in case of unauthorized attempts.
- **Sound Alarm**: A built-in buzzer module generates an audible alarm to alert the owner of suspicious activity.

The system architecture is modular, comprising submodules dedicated to distinct functionalities. This modular design facilitates easy maintenance, scalability, and future upgrades.

Furthermore, the project demonstrates an effective use of VHDL for state machine control, clock management, and peripheral integration. The design showcases robust system reliability and offers an excellent balance between security and usability. Overall, this smart combination lock serves as a practical and secure solution for safeguarding personal property.

Finally, future enhancements to this project may include:

- Adding a touchscreen or keypad interface for improved user interaction.
- Integrating a fingerprint sensor to enhance the security of the system.