University of Liège
Faculty of applied sciences

Project: A simple way to design a Catch The Light game

ELEN0040-1: Digital electronics

Group: 8

Authors :
Alyssa Di Matteo s201486
Catherine Duchemin s202046
Manon Gerard s201354
Neri Sansone s224179

Teacher :
J.-M. Redouté

Teaching assistants :
A. Fyon
A. Halin

Academic year 2022 - 2023

Introduction

This paper will explain the steps that need to be followed in order to design a simplified version of Catch The Light arcade game, represented in Figure 1. It can be played alone or in 1 vs 1, but we will only implement the solo part. If somebody wants to play 1 vs 1, it can be done by alternating the players and who gets the highest score wins.

FIGURE 1. Catch The Light arcade game

Let's explain the rules of the game. The game starts, when the player presses a button, turning on several lights. The player must press the buttons corresponding to the lighted LEDs during a limited amount of time in order to earn as many points as possible. When the player presses a lighted button, it turns off and a new LED is activated. If, on the other hand, the player presses a button corresponding to a non-illuminated LED, the player will lose some points as a penalty. The game ends when the timer expires.

Note that we simplified the rules of the real game. In reality, the lights turn on randomly and therefore there isn't a fixed number of illuminated LEDs.

In the following paragraph, we are going to outline the main steps needed to design this game on a CPLD using VHDL. The first step is to define the required hardware. Secondly, we will define the different inputs and outputs that the game needs. Then, the state machine diagram will be constructed. Afterwards, we will implement the VHDL code running the game, starting with the code for a specific state, then adding the other states to finally arrive at the complete code. Eventually, the construction of the game on the breadboard will be explained.

1 Required hardware

- 7 color LEDs : 6 red for the game , 1 green LED for reset
- 7 buttons and their associated resistances and capacitors in order to filter the button signal : 6 for the red LEDs, 1 for the reset LED
- 2 7-segment displays and 2 decoders : in order to display the score
- 3 breadboards

2 Inputs/Outputs of the Catch The Light arcade game

Concerning the inputs, we will use 2 pins for the 2 clocks of the development board, 6 pins for the different buttons that the player will have to press during the game and 1 pin for a reset button in order to start a new game when the previous game is over.

Concerning the outputs, we will use 6 pins for the 6 red LEDs and 1 pin for a specific LED indicating if the game timer has expired. Moreover, 8 pins will also be necessary for the 2 decoders of the 7-segment displays allowing to show the score.

The following input and output signals will be used in the game entity :

- `clock0` : input clock 0 signal coming from the 555 timer of the development board, this signal will be used for the random number generator
  $\longrightarrow$ *std_logic* ;
- `clock1` : input clock 1 signal coming from the 555 timer of the development board, this signal will be used as the main clock of the CPLD, and therefore will be used for the timer
  $\longrightarrow$ *std_logic* ;
- `buttons` : input signal of the 6 buttons associated with the 6 different LEDs of the game
  $\longrightarrow$ *std_logic_vector*(0 to 5) ;
- `reset` : input signal of the reset button to restart the game when the game is over
  $\longrightarrow$ *std_logic* ;
- `leds` : output signal of the 6 LEDs in order to turn on the light sequence
  $\longrightarrow$ *std_logic_vector*(0 to 5) ;
- `reset_led` : output signal of the reset LED in order to show whether the game is running or not
  $\longrightarrow$ *std_logic* ;
- `score1` and `score2` : the output signal of both digits is in binary, it will be after converted by the decoder, we will use `score1` for the units and `score2` for the tens.
  $\longrightarrow$ *integer* range 0 to 9 ;

The code of the whole entity is represented in Figure 2.

```
entity game_catch_the_light is port(
   clock0 : in std_logic;
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 5) := "000000";
   reset_led : out std_logic := '0';
   score1 : buffer integer range 0 to 9 := 0;
   score2 : buffer integer range 0 to 9 := 0;
   buttons : in std_logic_vector(0 to 5);
   reset : in std_logic
);
end entity game_catch_the_light;
```

Figure 2. Entity of the Catch The Light game

3 State machine diagram

Figure 3 represents the state machine diagram. The initial state of the game is **Add random LED** that adds a random number between 0 and 5 representing the LED to turn on. This state will be repeated until there are 3 (= max_lighten) LEDs on. The following state is **Display LED** whose role is to display the switched on LEDs. Then, if the timer has not expired yet, the player needs to push on the button associated to the illuminated LED. The state **Verification** is responsible for checking the timer. If the timer expires, the game goes in **End Game** state. However, if the timer is still running, the state **Verification** has to check the signal `button` and verify if it matches an illuminated LED. On one hand, if the user pushes on a non-illuminated LED, the game enters the **Penalty** state as he will lose a point. The game will immediately go back to the **Verification** state. On the other hand, if the user pushes on an illuminated LED, the following state will be **Switch off + reward**. After this, a random LED is switched on thanks to the state **Add random LED**. Eventually, the games goes in **End Game** state and when the reset button is pressed the new game will start from the state **Add random LED**.

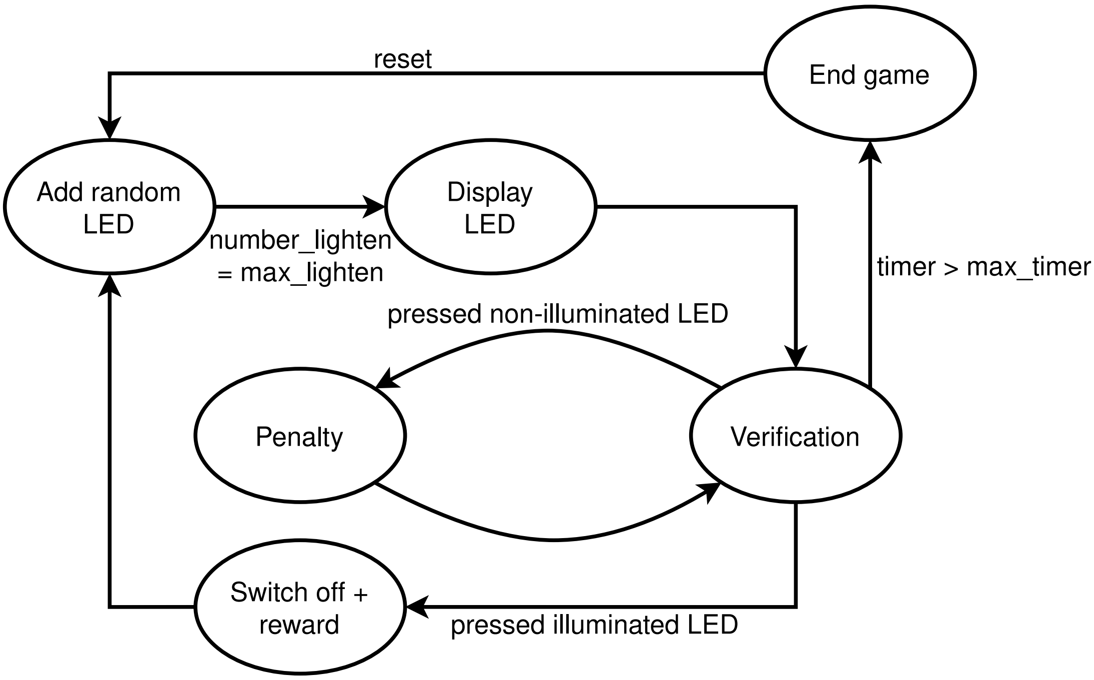

FIGURE 3. State machine of the Catch The Light game

From this, we can already think about essential signals and/or constant of the architecture of the game :

- `max_lighten` : a constant defining the maximum number of lighted LEDs
  ⟶ *integer* ;
- `lighten_seq` : a signal used to store the illuminated LED during the execution of the game, this will be a vector containing number ranging from 0 to 6. The usage of the number six will be explained later on in this paper.
  ⟶ *array*(0 to `number_lighten` - 1) of *integer* range 0 to 6 ;
- `state` : a signal used to indicate in which state the game currently is
  ⟶ state can have different types : add_random, display, verification, correct, wrong, end_game.

- `max_timer` : a constant defining the maximum value of the timer
  ⟶ *integer* ;
- `timer` : a signal indicating the current value of the timer during the game execution, its value is thus increased during the game
  ⟶ *integer* range from 0 to `max_timer` ;
- `random` : a signal used to have a random value
  ⟶ *integer* range 0 to 5 ;
- `new_user_command` : a signal which is true if there was no button pressed on the previous `clock1` tick
  ⟶ *boolean* ;

We will also use variables in the architecture of the game :

- `number_lighten` : a variable to indicate the number of illuminated LED during this state execution.
  ⟶ *integer* range from 0 to `max_lighten` ;
- `user_selection` : a variable indicating the index of the button pressed by the user
  ⟶ *integer* range from 0 to 5 ;
- `correct_button` : a variable to store whether the player pressed the button corresponding to an illuminated LED or not. This variable will later on be grouped with `non_illuminated`
  ⟶ *boolean* ;
- `non_illuminated` : a variable to store whether the random value picked is already an illuminated light or not
  ⟶ *boolean* ;

4 Codes of the different states

In order to avoid mistakes and difficulties, we decided to implement our game step by step. We first designed the **Display LED** state, then the **Verification** state. After this, we implemented the **Add random LED** state followed by the **Penalty** state and the **Switch off + reward** state. The **End game** state was the last one to be implemented.

On the CPLD, the `clock0` is the fastest one with a frequency range that goes from 59Hz to 320Hz. As a result, it will be used to generate a random number. Given that the `clock1` is slower (0,7Hz - 48,1Hz), it will be used as main clock of our game and it will compute the buttons as well as the LEDs.

4.1 Display LED state

This state must display the `max_lighten` illuminated LED contained in the `lighten_seq` signal using the output signal `leds`. To test this, we set `lighten_seq` as a constant since no random number generator is implemented yet, `leds` is set to zero initially and set back in state **End game**.

The sequence display will be performed as following :

1. The reset LED is off at start so `reset_led` is set to zero ;
2. The variable `i` is set to 0 ;

3. The LED of the `lighten_seq` signal at index `i` is turned on ;
4. `i` is incremented ;
5. The code goes back to step 3 until `i` has reached `max_lighten` - 1, meaning until there are `max_lighten` LEDs illuminated. When the code reaches `i` = `max_lighten`, the state **Display LED** is over.

An example of entity/architecture header is represented in Figure 4. An example of architecture code is represented in Figure 5.

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

entity display_LED is port(
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 5) := "000000";
   reset_led : out std_logic := '0'
);
end entity display_LED;

architecture display_LED_architecture of display_LED is

   constant max_lighten : integer := 3;

   type lighten_type is array(0 to max_lighten-1) of integer range 0 to 5;
   constant lighten_seq : lighten_type := (1, 0, 5);

   type states is (display, end_game);
   signal state : states := display;

begin
   state_machine : process(clock1)
```

FIGURE 4. Entity/architecture header of the state **Display sequence**

```
begin
   state_machine : process(clock1)
   begin
      if (rising_edge(clock1)) then
         case state is
            when display =>
               reset_led <= '0'; --The reset LED is off to indicate the game is running

               for i in 0 to max_lighten-1 loop
                  leds(lighten_seq(i)) <= '1'; --The LED of the lighten_seq at index i is on
               end loop;
               state <= end_game; --End of the sequence

            when end_game =>
               leds <= "000000"; --All the LEDs are off to indicate end of display
               reset_led <= '1'; --Except the reset LEDs to indicate the game can be started again

         end case;
      end if;
   end process state_machine;
end architecture display_LED_architecture;
```

FIGURE 5. Architecture of the state **Display sequence**

4.2 Verification state

After implementing the display of the lightened sequence, we now need to understand if the player has pushed a button that corresponds to an illuminated LED or not. We first need to know which button the user has pressed by checking the input signal `buttons`. In order to prevent this signal from being interpreted as a multiple selection of the same button, we set a new *boolean* signal `new_user_command` to true if there was no button pressed on the previous `clock1` tick. Thus, only the first falling edge of the signal from the button is interpreted as a new user command.

We use the following convention for the input signal `buttons` :

- 0V input is equal to '1' when pressing the button
- 5V input is equal to '0' when the button is unpressed

We decided to implement a filter to prevent bounces of the buttons. In fact, this isn't a fatal flaw as the bounces would be faster than the period of `clock1`. But we opted for a more general design. In fact, our implementation could also be used with a clock that has a higher frequency, for which the filter would be required.

In addition to that, we added an *integer* variable `user_selection` ranging from 0 to 5 containing the index of the button pressed by the user and a *boolean* variable `correct_button` to indicate whether the player pressed a button corresponding to an illuminated LED or not. We defined them as variables since they must be updated directly in order to be reused during the current execution of the process. Indeed, if the user presses 2 buttons during the same clock cycle, only the one with the highest index will be computed. We made this choice in order to respect our state diagram i.e. if we press two buttons and one is not correct we would have to go in two different states.

We also added two more variables : an *integer* constant `max_timer` and a `timer`. Their implementation will be changed later, but for the moment `timer` is increased in verification state and we leave that state when `timer` reaches `max_timer`. In the first place, this constant is set to 200, but eventually we will change it.

An example of updated entity/architecture header is represented in Figure 6.

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

entity verification_LED is port(
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 5) := "000000";
   reset_led : out std_logic := '0';
   buttons : in std_logic_vector(0 to 5)
);
end entity verification_LED;

architecture verification_LED_architecture of verification_LED is

   constant max_lighten : integer := 3;

   type lighten_type is array(0 to max_lighten-1) of integer range 0 to 5;
   constant lighten_seq : lighten_type := (1, 0, 5);

   type states is (display, verification, correct, wrong, end_game);
   signal state : states := display;

   signal new_user_command : boolean := false;

   constant max_timer : integer := 200;

begin
   state_machine : process(clock1)
   variable user_selection : integer range 0 to 5;
   variable correct_button : boolean := false;
   variable timer : integer range 0 to max_timer := 0;
   begin
      if (rising_edge(clock1)) then
```

FIGURE 6. Updated entity/architecture header

An example of updated state machine process is represented in Figure 7.

```
begin
   state_machine : process(clock1)
   variable user_selection : integer range 0 to 5;
   variable correct_button : boolean := false;
   variable timer : integer range 0 to max_timer := 0;
   begin
      if (rising_edge(clock1)) then
         case state is
            when display =>
               reset_led <= '0'; --The reset LED is off to indicate the game is running

               for i in 0 to max_lighten-1 loop
                  leds(lighten_seq(i)) <= '1'; --The LED of the lighten_seq at index i is on
               end loop;

               state <= verification; --End of the sequence

            when verification =>
               timer := timer + 1;
               if (timer = max_timer) then --Verify the timer
                  state <= end_game;

               elsif (buttons = "111111") then --No button pressed by the player
                  new_user_command <= true;
               elsif (new_user_command) then --New signal send by the player
                  new_user_command <= false;
                  for i in 0 to 5 loop
                     if (buttons(i) = '0') then
                        user_selection := i;
                     end if;
                  end loop; --Button signals interpretation

                  for i in 0 to max_lighten-1 loop
                     if (user_selection = lighten_seq(i)) then
                        correct_button := true;
                     end if;
                  end loop; --Verification of the chosen LED

                  if (correct_button = true) then --The player is correct
                     state <= correct;
                  else --The player is wrong
                     state <= wrong;
                  end if;
               end if;

            when correct =>
               state <= end_game;

            when wrong =>
               state <= end_game;

            when end_game =>
               leds <= "000000"; --All the LEDs are off to indicate end of display
               reset_led <= '1'; --Except the reset LEDs to indicate the game can be started again

         end case;
      end if;
   end process state_machine;
end architecture verification_LED_architecture;
```

FIGURE 7. Updated state machine process

4.3 Add random LED state

Now that we can display and verify a random sequence, it is time to choose the illuminated LED dynamically and randomly until the maximum number of illuminated LEDs (`max_lighten`) is reached. To do so, we need to implement a random number generator.

In VHDL, there is no direct way to create a random number as the result of the logic circuit is deterministic. However, it is possible to create an unknown signal for the player but still deterministic for the circuit. This is called pseudo-randomness. To do so, an external input which is unpredictable is necessary. The one we chose is the time at which the player will press a button. This will therefore impact the clock cycle at which the random value is in the process of `clock0`. A few cycle after this input, the random number will be used. This `random` number takes the form of an *integer* signal varying from 0 to 5 on all the rising edges of `clock0`. As the `random_generator` process and the `state_machine` process are triggered by two different clocks, the value read in the `state_machine` and written in the `random_generator` will be random. This two-step process is requested because assigning a signal to two different processes could produce a voltage conflict between two output voltages.

We also need to check that the chosen LED is not already lighten. If that is the case, we will not leave the current state until the chosen LED is not already lighted. Therefore, it is required to add a *boolean* variable `non_illuminated` to understand whether or not the picked LED is illuminated. By default, it is set to true in this stage and turned to false when needed. In addition to that, we will also use the `number_lighten` variable to indicate the number of illuminated LED during this state execution. It is an *integer* that can go from 0 to `max_lighten`. The variable `number_lighten` is set at zero initially and set back in state **End game**. They must be defined as variables since they must be updated directly in order to be reused during the current execution of the process.

The state Add a random LED will follow these steps :

1. The reset LED is off at start so `reset_led` is set at zero.
   This was previously considered to be in the display state ;
2. The variable `non_illuminated` is set to true ;
3. There is a comparison between the LED of the `lighten_seq` signal at each index and the `random` signal ;
4. If at least one is equal, the variable `non_illuminated` is set to false ;
5. If `non_illuminated` is true, the random number is added in the `lighten_seq` at index `number_lighten`, which is afterward increased ;
6. When the code reaches `number_lighten` = `max_lighten`, the state **Add random LED** is over. And otherwise, it starts again at step 1

Sometimes during the game, or when the game starts, there can be less LEDs lighten on than the `max_lighten` value. Therefore, we adopted the convention of using the number 6 for the "missing LEDs" when the sequence is not full. Indeed values from 0 to 5 were already taken for the 6 red LEDs. Adding this number 6 is fine because representing 6 or 7 possible values takes the same amount of bits. As a result, `lighten_seq` is set at (6, 6, 6) initially and set back to (6, 6, 6) in state **End game**.

An example of updated entity/architecture header and random generator process is represented in Figure 8

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

entity add_random_LED is port(
   clock0 : in std_logic;
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 5) := "000000";
   reset_led : out std_logic := '0';
   buttons : in std_logic_vector(0 to 5)
);
end entity add_random_LED;

architecture add_random_LED_architecture of add_random_LED is

   constant max_lighten : integer := 3;

   type lighten_type is array(0 to max_lighten-1) of integer range 0 to 6;
   signal lighten_seq : lighten_type := (6, 6, 6);

   type states is (add_random, display, verification, correct, wrong, end_game);
   signal state : states := add_random;

   signal new_user_command : boolean := false;

   constant max_timer : integer := 200;

   signal random : integer range 0 to 5;

begin
   random_generator : process(clock0)
   begin
      if (rising_edge(clock0)) then
         if (random = 5) then
            random <= 0;
         else
            random <= random + 1;
         end if;
      end if;
   end process random_generator;
```

FIGURE 8. Updated entity/architecture header with the random generator process

An example of updated state machine process is represented in Figure 9.

```
state_machine : process(clock1)
variable number_lighten : integer range 0 to max_lighten := 0;
variable user_selection : integer range 0 to 5;
variable correct_button : boolean := false;
variable timer : integer range 0 to max_timer := 0;
variable non_illuminated : boolean := true;
begin
   if (rising_edge(clock1)) then
      case state is
         when add_random =>
            reset_led <= '0'; --The reset LED is off to indicate the game is running

            non_illuminated := true;
            if (lighten_seq(0) = random or lighten_seq(1) = random or lighten_seq(2) = random)
               non_illuminated := false;
            end if; --Verification whether the LED is non-illuminated

            if (non_illuminated = true) then
               --The random number is added to the lighten LEDs
               lighten_seq(number_lighten) <= random;
               number_lighten := number_lighten + 1; --The size of the sequence is incremented
            end if;

            if (number_lighten = max_lighten) then
               state <= display;
            end if;
```

```vhdl
            when display =>
               for i in 0 to max_lighten-1 loop
                  leds(lighten_seq(i)) <= '1'; --The LED of the lighten_seq at index i is on
               end loop;
               state <= verification; --End of the sequence

            when verification =>
               timer := timer + 1;
               if (timer = max_timer) then --Verify the timer
                  state <= end_game;

               elsif (buttons = "111111" ) then --No button pressed by the player
                  new_user_command <= true;
               elsif (new_user_command ) then --New signal send by the player
                  new_user_command <= false;
                  for i in 0 to 5 loop
                     if (buttons(i) = '0') then
                        user_selection := i;
                     end if;
                  end loop; --Button signals interpretation

                  correct_button := false;
                  for i in 0 to max_lighten-1 loop
                     if (user_selection = lighten_seq(i)) then
                        correct_button := true;
                     end if;
                  end loop; --Verification of the chosen LED

                  if (correct_button = true) then --The player is correct
                     state <= correct;
                  else --The player is wrong
                     state <= wrong;
                  end if;
               end if;

            when correct =>
               state <= end_game;

            when wrong =>
               state <= end_game;

            when end_game =>
               number_lighten := 0;
               leds <= "000000"; --All the LEDs are off to indicate end of display
               lighten_seq <= (6, 6, 6); --Reset the lighten sequence
               reset_led <= '1'; --Except the reset LEDs to indicate the game can be started again

         end case;
      end if;
   end process state_machine;
end architecture add_random_LED_architecture;
```

FIGURE 9. Updated state machine process

4.4 End game

The following step is to code the **End game** state. We designed it to switch off all the LEDs once the game is ended, except for the reset one, that on the contrary, will be lighted up. In this state, it is necessary to wait that the user presses the reset button to go back to the **Add random LED** state. The activation of the button also resets the `timer` and `reset_led` to 0.

An example of architecture of the **End game** state is represented in Figure 10.

```
        when end_game =>
           number_lighten := 0;
           leds <= "000000"; --All the LEDs are off to indicate end of display
           lighten_seq <= (6, 6, 6); --Reset the lighten sequence
           reset_led <= '1'; --Except the reset LEDs to indicate the game can be started again
           if (reset = '0') then
              state <= add_random;
              reset_led <= '0'; --The reset LED is off to indicate the game is running
              timer := 0;
           end if;

      end case;
   end if;
  end process state_machine;
end architecture end_of_game_architecture;
```

FIGURE 10. Architecture of the **End game** state

4.5 Penalty and Switch off + reward state

We need to implement the **Penalty** and **Switch off + reward** states. It is at this stage that we add the `score`. In addition, we should modify **End game** so that the 7-segment displays still show the score of the game once it is finished. This score will be reset at the same time as the game is reset. In order to distinguish the units from the tens, we use 2 different buffers `score1` and `score2` in our code.

When we are in the **Penalty** state, the score decreases by one. On the contrary, in the **Switch off + reward** state, the score increases by one. In addition, in this state, the LED corresponding to the button pushed should be switched off. Therefore, we added a signal `index_correct`. It is an *integer* that can go from 0 to `max_lighten`-1. Its value is given in the verification state where in addition to stating if the button is the correct, its index is stored in `lighten_seq`. In **Switch off + reward** state, the LED can therefore be set to 0. Afterward, the sequence is shifted in order to take out the switched off LED and `number_lighten` is decreased. This shifting operation is needed so that the **Add random LED** state does not need to be adapted. Note that, the last position of `lighten_seq` stays equal to its value because it won't be considered anymore and that it is equal to the penultimate value so that will not cause any problem in the **Add random** state. It would also have been possible to set the last position of `lighten_seq` to our convention 6, but this would have required a useless operation.

Note that for simplification, we said that the score increases and decreases, but as we had to separate in two different signals the units and the tens the code had to take this into account.

An example of updated entity/architecture header is represented in Figure 11.

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

entity penalty_reward is port(
   clock0 : in std_logic;
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 5) := "000000";
   reset_led : out std_logic := '0';
   score1 : buffer integer range 0 to 9 := 0;
   score2 : buffer integer range 0 to 9 := 0;
   buttons : in std_logic_vector(0 to 5);
   reset : in std_logic
);
end entity penalty_reward;

architecture penalty_reward_architecture of penalty_reward is

   constant max_lighten : integer := 3;

   type lighten_type is array(0 to max_lighten-1) of integer range 0 to 6;
   signal lighten_seq : lighten_type := (6, 6, 6);

   type states is (add_random, display, verification, correct, wrong, end_game);
   signal state : states := add_random;

   signal new_user_command : boolean := false;
   signal index_correct : integer range 0 to max_lighten-1;

   constant max_timer : integer := 200;

   signal random : integer range 0 to 5;

begin
   random_generator : process(clock0)
   begin
      if (rising_edge(clock0)) then
         if (random = 5) then
            random <= 0;
         else
            random <= random + 1;
         end if;
      end if;
   end process random_generator;
```

FIGURE 11. Updated entity/architecture header

An example of updated state machine process is represented in Figure 12.

```
state_machine : process(clock1)
variable number_lighten : integer range 0 to max_lighten := 0;
variable user_selection : integer range 0 to 5;
variable correct_button : boolean := false;
variable timer : integer range 0 to max_timer := 0;
variable non_illuminated : boolean := true;
begin
   if (rising_edge(clock1)) then
      case state is
         when add_random =>
            non_illuminated := true;
            if (lighten_seq(0) = random or lighten_seq(1) = random or lighten_seq(2) = random) then
               non_illuminated := false;
            end if; --Verification whether the LED is non-illuminated

            if (non_illuminated = true) then
               --The random number is added to the lighten LEDs
               lighten_seq(number_lighten) <= random;
               number_lighten := number_lighten + 1; --The size of the sequence is incremented
            end if;

            if (number_lighten = max_lighten) then
               state <= display;
            end if;
```

```vhdl
        when display =>
           for i in 0 to max_lighten-1 loop
              leds(lighten_seq(i)) <= '1'; --The LED of the lighten_seq at index i is on
           end loop;
           state <= verification; --End of the sequence

        when verification =>
           timer := timer + 1;
           if (timer = max_timer) then --Verify the timer
              state <= end_game;

           elsif (buttons = "111111") then --No button pressed by the player
              new_user_command <= true;
           elsif (new_user_command) then --New signal send by the player
              new_user_command <= false;
              for i in 0 to 5 loop
                 if (buttons(i) = '0') then
                    user_selection := i;
                 end if;
              end loop; --Button signals interpretation

              correct_button := false;
              for i in 0 to max_lighten-1 loop
                 if (user_selection = lighten_seq(i)) then
                    correct_button := true;
                    index_correct <= i;
                 end if;
              end loop; --Verification of the chosen LED

              if (correct_button = true) then --The player is correct
                 state <= correct;
              else --The player is wrong
                 state <= wrong;
              end if;
           end if;

        when correct =>
           leds(lighten_seq(index_correct)) <= '0';
           for k in 0 to max_lighten-2 loop
              if (k >= index_correct) then
                 lighten_seq(k) <= lighten_seq(k+1);
              end if;
           end loop; --Switch off
           number_lighten := number_lighten - 1;

           if (score1 = 9) then
              if (score2 < 9) then
                 score1 <= 0;
                 score2 <= score2 + 1;
              end if;
           else
              score1 <= score1 + 1;
           end if; --Reward
           state <= add_random;

        when wrong =>
           if (score1 = 0) then
              if (score2 >= 1) then
                 score1 <= 9;
                 score2 <= score2 - 1;
              end if;
           else
              score1 <= score1 - 1;
           end if; --Penalty
           state <= verification;

        when end_game =>
           number_lighten := 0;
           leds <= "000000"; --All the LEDs are off to indicate end of display
           lighten_seq <= (6, 6, 6); --Reset the lighten sequence
           reset_led <= '1';  --Except the reset LEDs to indicate the game can be started agair
           if (reset = '0') then
              score <= 0; --reset the score that was still displayed at the end of the game
              state <= add_random;
              reset_led <= '0'; --The reset LED is off to indicate the game is running
              timer := 0;
           end if;

      end case;
    end if;
  end process state_machine;
end architecture penalty_reward_architecture;
```

FIGURE 12. Updated state machine process

4.6 Timer + Optimization

Finally, we implemented a timer. This timer will be incremented at each clock cycle of `clock1`, our main CPLD clock, `timer` can now become a signal. When `timer` becomes equal to `max_timer` this corresponds to our expression "timer > max_timer" in the state machine diagram. This is like this because `timer` is a signal and not a variable, so it is updated at the end of the clock cycle. Therefore, when `timer` = `max_timer`, it means that we reached our `max_timer` at the previous clock cycle. Note that, VHDL needs the maximum value that the signal can reach to compute the number of bits needed to represent the signal. Therefore, we limited the maximum value of `timer` to `max_timer` as our condition is for when they are both equal, so there is no need to be able to reach a higher value. When this condition is tested and met in the **Verification** state, the next state executed is **End game**.

At first, we decided to set the frequency of `clock1` to its highest value. Turning the `clock1` button allows us to increase (resp. decrease) the code execution. Thus, the playing time decreases (resp. increases) and the game gets harder (resp. easier). However, we can not set a clock frequency to low, otherwise it will not detect the button activation. On the other hand, a higher clock could lead to a small amount of playing time which could make the game unplayable, so we have to set our `max_timer` to a decent value.

Secondly, we decided that the perfect gameplay should last around 20 secs. In order to do so we set the value of `max_timer` to 1000 clock cycles which brings the playing time up to 17 seconds using a moderate range of bits, which is a good compromise.

In order to optimize a little bit our code, we realized we could group the two variables `non_illuminated` and `correct_button` into a single variable `non_illuminated`. This can be done because their values are needed only if one state is different from each other. In order for our code to be comprehensible, `correct_button` which becomes `non_illuminated` has the contrary signification, i.e. the value true for `correct_button` becomes false, and vice versa.

The final entity/architecture header is represented in Figure 13.

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

entity game_catch_the_light is port(
   clock0 : in std_logic;
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 5) := "000000";
   reset_led : out std_logic := '0';
   score1 : buffer integer range 0 to 9 := 0;
   score2 : buffer integer range 0 to 9 := 0;
   buttons : in std_logic_vector(0 to 5);
   reset : in std_logic
);
end entity game_catch_the_light;

architecture game_catch_the_light_architecture of game_catch_the_light is

   constant max_lighten : integer := 3;

   type lighten_type is array(0 to max_lighten-1) of integer range 0 to 6;
   signal lighten_seq : lighten_type := (6, 6, 6);

   type states is (add_random, display, verification, correct, wrong, end_game);
   signal state : states := add_random;

   signal new_user_command : boolean := false;
   signal index_correct : integer range 0 to max_lighten-1;

   constant max_timer : integer := 1000;
   signal timer : integer range 0 to max_timer := 0;

   signal random : integer range 0 to 5;
```

FIGURE 13. Entity/architecture header of the Catch The Light game

The final architecture code is represented in Figure 14.

```
state_machine : process(clock1)
variable number_lighten : integer range 0 to max_lighten := 0;
variable user_selection : integer range 0 to 5;
variable non_illuminated : boolean := true;
begin
   if (rising_edge(clock1)) then
      if (timer < max_timer) then --Increase the timer at each clock cycle
         timer <= timer + 1;
      end if;
      case state is
         when add_random =>
            non_illuminated := true;
            if (lighten_seq(0) = random or lighten_seq(1) = random or lighten_seq(2) = random) then
               non_illuminated := false;
            end if; --Verification whether the LED is non-illuminated

            if (non_illuminated = true) then
               --The random number is added to the lighten LEDs
               lighten_seq(number_lighten) <= random;
               number_lighten := number_lighten + 1; --The size of the sequence is increamented
            end if;

            if (number_lighten = max_lighten) then
               state <= display;
            end if;

         when display =>
            for i in 0 to max_lighten-1 loop
               leds(lighten_seq(i)) <= '1'; --The LED of the lighten_seq at index i is on
            end loop;
            state <= verification; --End of the sequence
```

```
         when verification =>
            if (timer = max_timer) then --Verify the timer
               state <= end_game;

            elsif (buttons = "111111") then --No button pressed by the player
               new_user_command <= true;
            elsif (new_user_command) then --New signal send by the player
               new_user_command <= false;
               for i in 0 to 5 loop
                  if (buttons(i) = '0') then
                     user_selection := i;
                  end if;
               end loop; --Button signals interpretation

               non_illuminated := true;
               for i in 0 to max_lighten-1 loop
                  if (user_selection = lighten_seq(i)) then
                     non_illuminated := false;
                     index_correct <= i;
                  end if;
               end loop; --Verification of the chosen LED

               if (non_illuminated = false) then --The player is correct
                  state <= correct;
               else --The player is wrong
                  state <= wrong;
               end if;
            end if;

         when correct =>
            leds(lighten_seq(index_correct)) <= '0';
            for k in 0 to max_lighten-2 loop
               if (k >= index_correct) then
                  lighten_seq(k) <= lighten_seq(k+1);
               end if;
            end loop; --Switch off
            number_lighten := number_lighten - 1;

            if (score1 = 9) then
               if (score2 < 9) then
                  score1 <= 0;
                  score2 <= score2 + 1;
               end if;
            else
               score1 <= score1 + 1;
            end if; --Reward
            state <= add_random;

         when wrong =>
            if (score1 = 0) then
               if (score2 >= 1) then
                  score1 <= 9;
                  score2 <= score2 - 1;
               end if;
            else
               score1 <= score1 - 1;
            end if; --Penalty
            state <= verification;

         when end_game =>
            number_lighten := 0;
            lighten_seq <= (6, 6, 6); --Reset the lighten sequence
            leds <= "000000"; --All the LEDs are off to indicate end of display
            reset_led <= '1';  --Except the reset LEDs to indicate the game can be started agair
            if (reset = '0') then
               score1 <= 0; --reset the score that was still displayed at the end of the game
               score2 <= 0;
               state <= add_random;
               reset_led <= '0'; --The reset LED is off to indicate the game is running
               timer <= 0;
            end if;

      end case;
   end if;
  end process state_machine;
end architecture game_catch_the_light_architecture;
```

Figure 14. Architecture of the Catch The Light game

5 Tests

The VHDL code was tested with a testbench. To facilitate the comprehension, three signals `testTimer`, `testRandom` and `testState` were added in the entity. `testTimer` and `testRandom` are equal to the signals `timer` and `random`. `testState` is an integer that is set to a different value depending on the state. This allowed us to verify that the states would change correctly.

Here are the execution of the different cycles that can be observed in Figure 15 :

Cycle	State	Description
1-3	`add_random`	fill in `lighten_leq` with the value of `random` (3, 4 and 5)
4	`display`	the LEDs in `lighten_leq` are switch on
5	`verification`	`max_timer` not reached and no button pressed $\rightarrow$ `new_user_command` becomes true
6	`verification`	5^{th} button pressed, so verification $\rightarrow$ `correct_button` becomes true and the index is stored in `index_correct`
7	`correct`	increase the `score1` and switch off the 5^{th} LED
8	`add_random`	fill in `lighten_leq` with the value of `random` (2)
9	`display`	the 2^{nd} LEDs is switch on
10-11	`verification`	`max_timer` not reached and no button pressed $\rightarrow$ `new_user_command` becomes true
12	`verification`	1^{st} button pressed, so verification $\rightarrow$ `correct_button` becomes false
13	`wrong`	decrease the `score1`
14-16	`verification`	`max_timer` not reached and no button pressed $\rightarrow$ `new_user_command` becomes true
17	`verification`	6^{th} button pressed, so verification $\rightarrow$ `correct_button` becomes true and the index is stored in `index_correct`
18	`correct`	increase the `score` and switch off the 6^{th} LED
19	`add_random`	the LED at random (3) is already switch on
20	`add_random`	fill in `lighten_leq` with the value of `random` (4)
21	`display`	the 4^{th} LEDs is switch on
22-25	`verification`	`max_timer` not reached and no button pressed $\rightarrow$ `new_user_command` becomes true
26	`verification`	`max_timer` reached : meaning in reality > `max_timer` Ok, we are at the 26^{th} cycle
27-29	`end_game`	switch off all leds and turn on `reset_led`, still display the score
30	`end_game`	`reset` pressed $\rightarrow$ `reset_led` is switched off and `score` and `timer` becomes 0
...	...	starts again correctly

Note that, at each cycle of `clock1` the timer is increased, but we did not mention it in the table. The testbench was realized with `clock0` set with a period of 3ms and `clock1` of 20ms. These frequencies are not the same as the ones in our CPLD, but the code still works for our values.

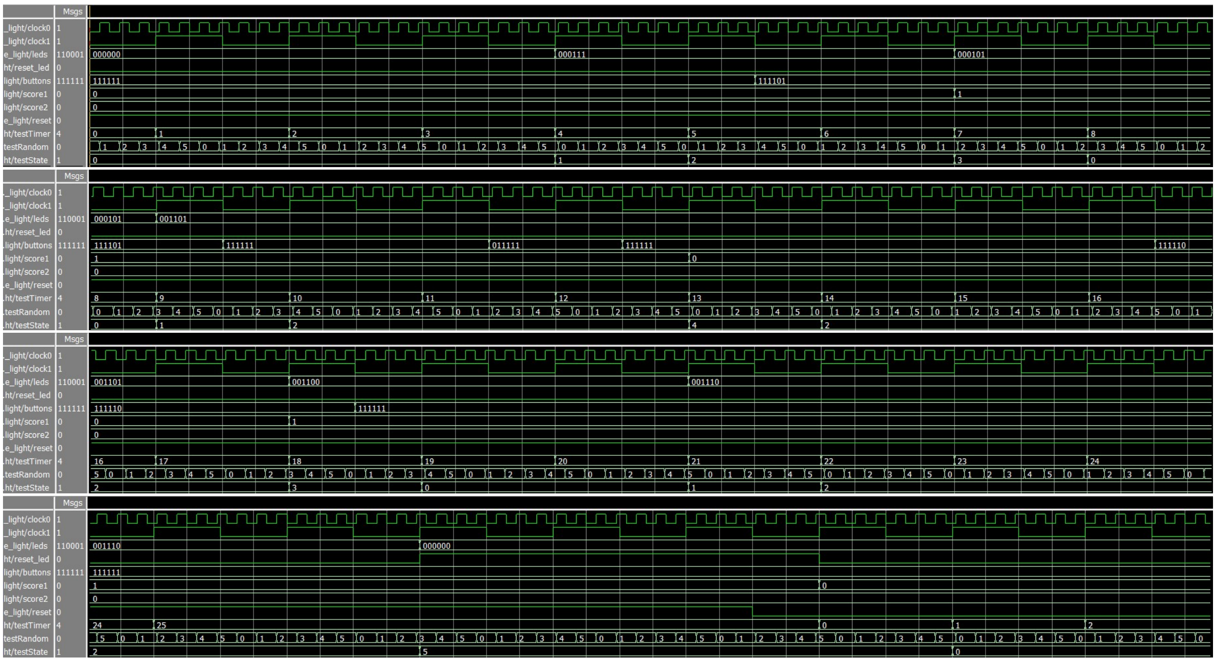

Figure 15. Testbench of the Catch The Light game

6 Construction

The final construction of our game on breadboards is represented in Figure 16.

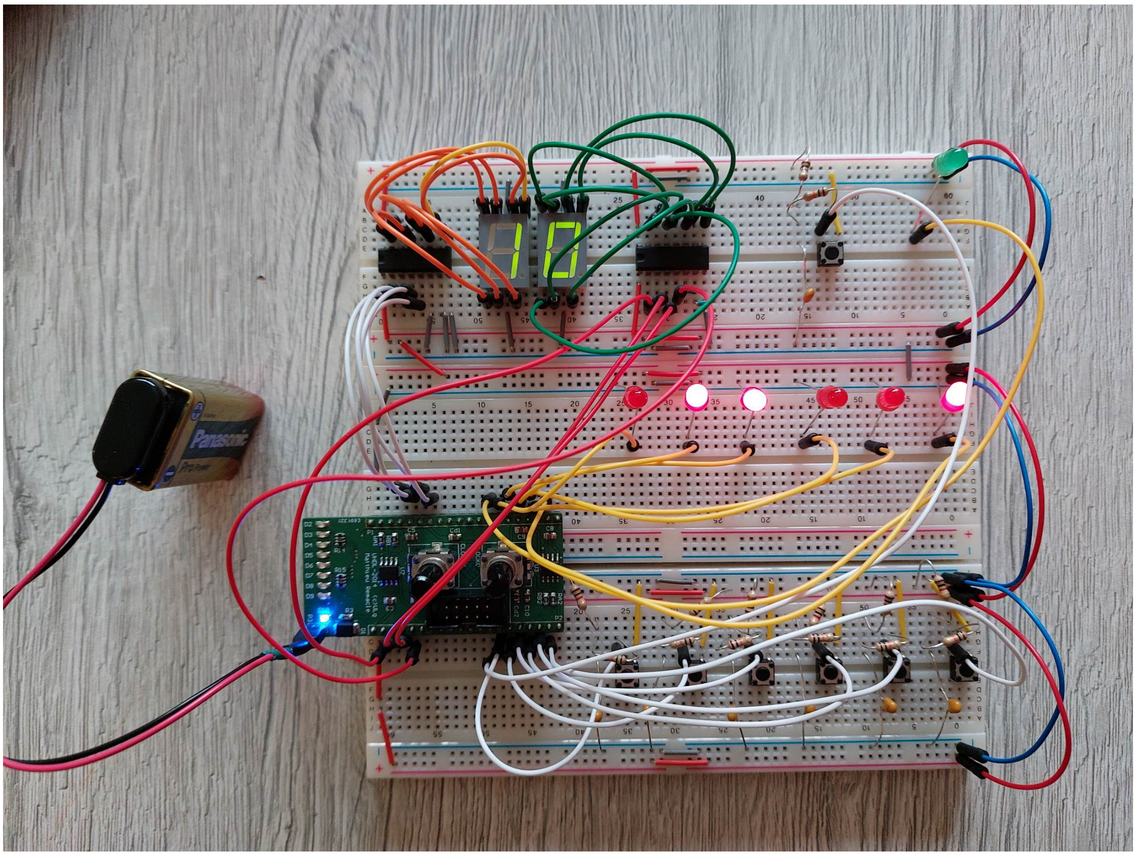

Figure 16. Construction of Catch The Light game on breadboards

The CPLD pin assignment is presented in the following table.

Pin assignment		
Node Name	Location	CPLD Pin
clock0	PIN_7	
clock1	PIN_9	
reset_led	PIN_56	P1_13
leds[0]	PIN_58	P1_14
leds[1]	PIN_59	P1_15
leds[2]	PIN_60	P1_16
leds[3]	PIN_61	P1_17
leds[4]	PIN_62	P1_18
leds[5]	PIN_63	P1_19
reset	PIN_12	P1_8
buttons[0]	PIN_11	P2_7
buttons[1]	PIN_5	P2_6
buttons[2]	PIN_4	P2_5
buttons[3]	PIN_3	P2_4
buttons[4]	PIN_2	P2_3
buttons[5]	PIN_1	P2_2
score1[0]	PIN_22	P2_16
score1[1]	PIN_25	P2_18
score1[2]	PIN_26	P2_19
score1[3]	PIN_24	P2_17
score2[0]	PIN_47	P1_7
score2[1]	PIN_45	P1_5
score2[2]	PIN_44	P1_4
score2[3]	PIN_46	P1_6

In addition to the test-bench, while building our circuit, we tested that : each button, LEDs, decoder and 7-segment displays were well constructed. Then, we tested the different versions of our code that we implemented one state at a time.

The electronic schematic of our construction is represented in Figure 17 and Figure 18.

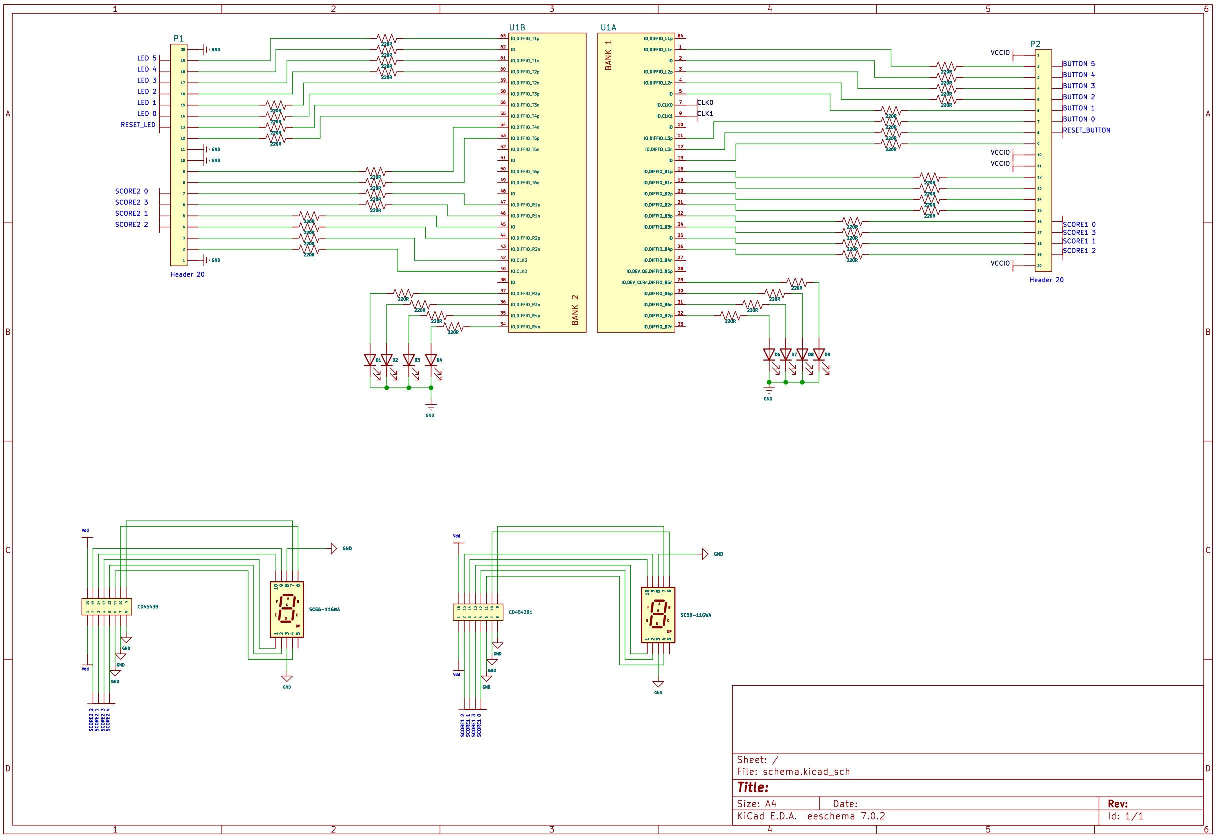

FIGURE 17. Electronic schema

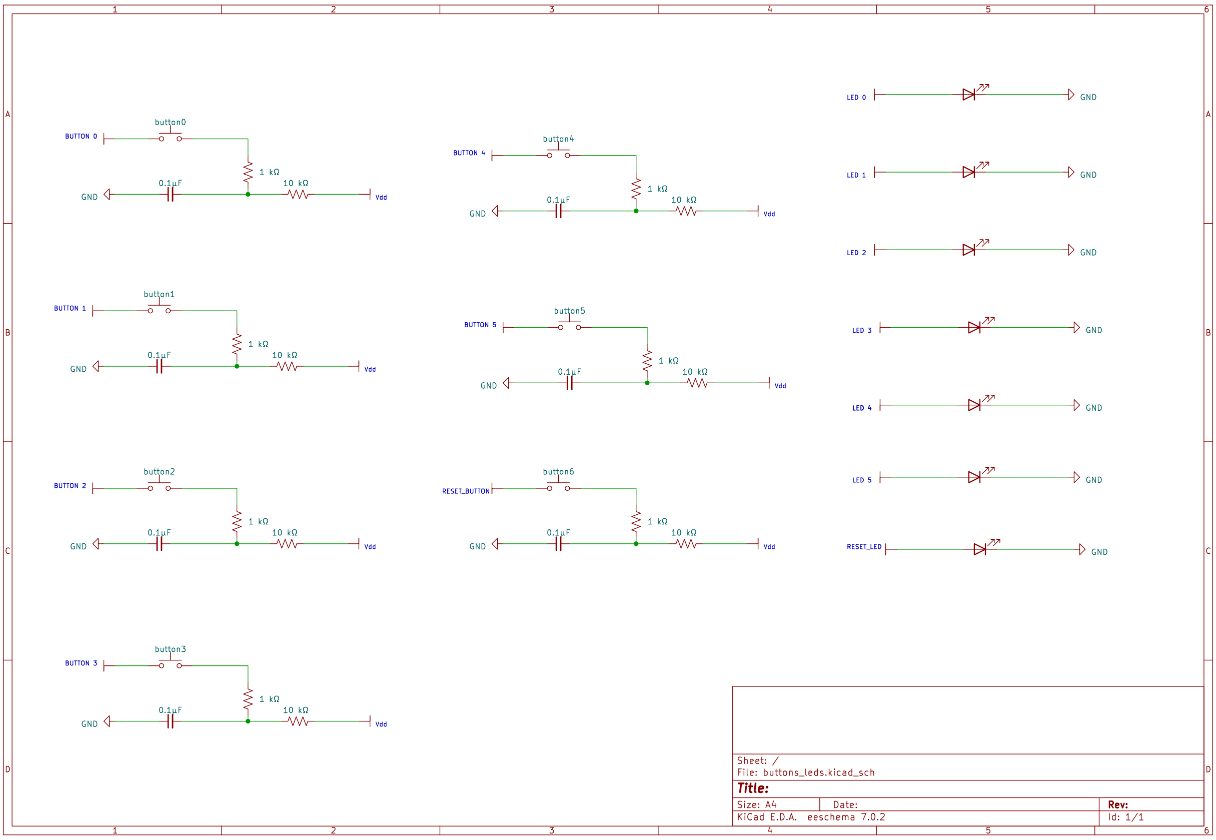

FIGURE 18. Electronic schema

7 Comments

While developing this project we tried to use as many signals as possible but it has not been always possible in fact sometimes we needed to update a signal multiple times during the same clock cycle, so we implemented some variables even though it is not optimal.

In fact, we tried to optimize the code by using the least amount of variables we could and therefore use the more signals we could. As mentioned in section 4.6, we grouped variables `non_illuminated` and `correct_button` into a single variable to optimize the code. The variables that are still present in the code could not become signals as we needed to dynamically update them.

Thirdly, note that the random generator could be improved. It is not real randomness as we mentioned earlier. Furthermore, in our implementation, we lose some clock cycles if the random index is equal to an already illuminated LED. This could have been enhanced by selecting the next value of random in the same clock cycle until we reach a LED that can be lightened up. But this also has its disadvantages as we would have needed an extra variable instead of a signal.

Eventually, the score is limited to values between 0 and 99 as we chose to have two 7-segment displays. In order to increase the value of the score, we would need a higher max score and therefore more 7-segment displays. It would also have been possible to implement the decoder for the 7-segment displays directly in VHDL, but we opted for IC decoders.

8 Conclusion

The final code of this Catch The Light arcade game is contained in the file *game_catch_the_light.vhd*.