DIGITAL ELECTRONICS
PROJECT 2:
SAFE WITH COMBINATION LOCK

ALEJANDRO LÓPEZ RODRÍGUEZ
ANA Mª CASANOVA LÓPEZ
RAÚL GÓMEZ FETES

Universidad Politécnica de Valencia
ETSID

MAY 8, 2020

Preface

This project is the result of weeks of collaborative work between the authors. Designing, discussing and testing the different modules that conform it has helped us to better understand the course as well as to develop crucial key skills such as communication, teamwork, problem solving and time management.

Before diving into the document there are some remarks that must be made to better understand the assignment:

- Most of the GALs that we have used in this project use Finite State Machines to perform their different duties. In almost of all them, a synchronous reset is used in lieu of an asynchronous one. The decision was intentional since most of the FSMs end in a final steady state that waits for the reset signal to restart themselves. Besides, the clock frequency is very fast for a GAL, so it is almost impossible for them to miss the reset signal. We have tested this extensively and found no problem whatsoever in the execution of the code.
- All of the images, tables, and figures are vectorial, so feel free to zoom in if something is not big enough.
- There are hyperlinks in most of the pages that will take you to the specific section in which the mentioned topic is explained in more detail. We recommend that you use a PDF Viewer such as Adobe Acrobat or similar.
- We have also added hyperlinks to the references that we have used to elaborate the assignment. The full references list can be found at the end of the document.
- The project files can be found **here**. We strongly suggest having the Proteus file open alongside the PDF.
- In the Proteus file you will find instructions on how to operate the system.

We sincerely hope you like the project. Please, feel free to comment on how you would improve certain aspects if you find it necessary. We are aware of the extension of the document and would like to apologize beforehand.

Finally, let us end this project and also this subject by saying that it has truly been a great pleasure having you as a teacher. We hope that somehow our paths cross again in the future.

Contents

Introduction

A security box or safe-deposit box stores our personal belongings safely, only being accessible by ourselves or the ones we trust. It is common to have one at home, but they can also be found at hotel rooms. This device is usually a metal box, often with a concrete layer for not only making it more secure, but heavier, so that burglars have it more difficult to take it with them. The door mechanism is usually a multi-point lock, and the opening method depends on the security box. Different opening mechanisms are widely used in commercial security boxes, namely a key, a multi-combination mechanism, a keypad, a fingerprint scanner, and so on.

1.1 Objectives

The goal of this project is to develop the whole electronics part of a security box. The security box will be operated by a keypad in order to enter a 4-digit password. To set the password that the user wants, dip switches will be installed inside, so that only the user can change it.

1.2 Opening sequence

To enter a password, the "start button" or asterisk * must be pressed first. Then, the password can be introduced, with two restrictions:

- Password must be a 4 digit number.
- Password must not have two or more consecutive repeated digits, i.e. 1122, 1111 or 1223 are not valid.

In order to open/close the security box, the "enter password" button or hash # must be pressed. Only then the password will be processed by the system. Once pressed, the password will appear on the screen for a brief amount of time.

After this, both passwords, the one entered by the keypad and the one stored inside the security box will be compared. If they are equal, after a few seconds an *Open* message will appear, and, at the same time, a green LED will be switched on, which in real life would be the motor to open the multi-point lock; otherwise a message of *Error* will be displayed and a red LED will be switched on, indicating that the passwords did not match.

1.3 Password setting mechanism

Once the security box is opened, the user has the option to set a new password. The mechanism involves 4 dip switches, one for each of the 4 digits.

Each digit must be in the range of 0 to 9 and must be introduced in binary, as each selector consists of four independent switches, being the top one the Most Significant Bit (MSB).

The new password must meet the requirements from Subsection 1.2, because otherwise, the GALs will not be able to compare both passwords correctly.

Implementation

2.1 Keypad

The most common method of introducing the password of a security box is the keypad. This is due to its ease of use, and its low cost, compared for instance with a dial combination, or a fingerprint scanner.

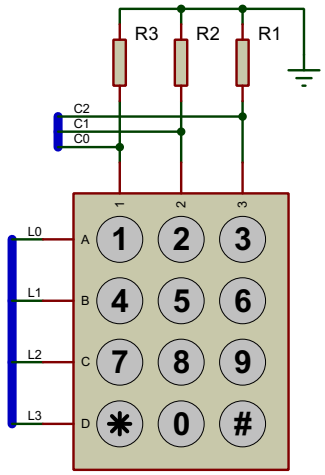


Figure 1: Keypad from Proteus' Libraries

In order to implement a keypad in our system, a HIGH level is introduced in one of the line rows and it is constantly circulated among them. When a button is pressed, depending on the column in which a HIGH level is detected and the position of the '1' in the rows, the GAL will be able to show in the output the Binary Coded Decimal (BCD) representation of the number that has been pressed following this fashion:

L3..L0	C2...C0	Button	BCD code
0001	001	1	0001
0010	001	4	0100
0100	001	7	0111
1000	001	*	1100
0001	010	2	0010
0010	010	5	0101
0100	010	8	1000
1000	010	0	0000
0001	100	3	0011
0010	100	6	0110
0100	100	9	1001
1000	100	#	1111

Table 1: Reading to Button Conversion table [1]

A ring counter is used in order to rotate the HIGH input bit for the lines. It is a type of counter composed of flip-flops connected together to form a shift register, with the output of the last flip-flop fed to the input of the first, making a "circular" or "ring" structure. In this case, the ring must be initialized with a '1' and after that, the bit will be rotating indefinitely. An example of such counter can be found attached below:

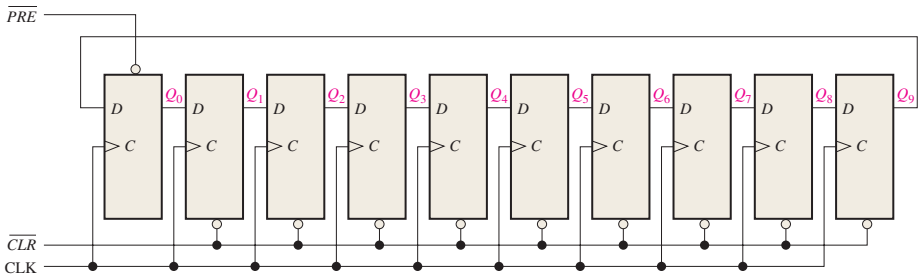


Figure 2: 10-bit Ring Counter [2]

The initialization of the ring counter is done with a pulse coming from the auto start timer that is fed into the *START* input of the GAL, see **Subsection 2.10** for more on this. Once this pulse is detected by the GAL, it introduces a bit in the *l_temp* internal signal starting the counter.

The *READY* signal is an output that is needed in order to produce the auto start of the system, more on this in **Subsubsection 2.10.2**.

Besides, we can also find an output 4 bit array, called *SIG*, which will output the BCD Code that we have previously discussed.

This I/O configuration can be seen in the following image:

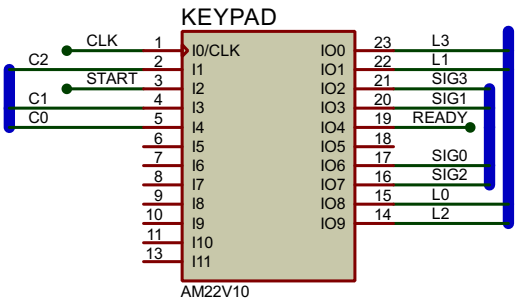


Figure 3: Proteus Subassembly of Keypad

The VHDL code that we used for this part of the system is very similar to the one that we used in the practical sessions. It can be found below:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY KEYPAD IS
  PORT (
    START, CLK : IN std_logic; --start pulse
    C : IN std_logic_vector(2 DOWNTO 0); --column of keypad
    L : OUT std_logic_vector(3 DOWNTO 0); -- lines of keypad
    SIG : OUT std_logic_vector(3 DOWNTO 0); -- BCD of key pressed
    pulse : OUT std_logic; --a button is pressed
    READY : OUT std_logic := '1'
  );
END KEYPAD;

ARCHITECTURE ARCH OF KEYPAD IS
  SIGNAL l_temp : std_logic_vector(3 DOWNTO 0); --temporal value line
BEGIN
  pulse <= (C(0) OR C(1) OR C(2)); --detects if a button is pressed
  PROCESS (CLK)
  BEGIN
    IF (CLK = '1' AND CLK'EVENT) THEN
      IF START = '1' THEN --start pulse
        l_temp <= "0001"; --bit for ring introduced
      ELSE --ring
        l_temp(1) <= l_temp(0);
        l_temp(2) <= l_temp(1);
        l_temp(3) <= l_temp(2);
        l_temp(0) <= l_temp(3);
      END IF;
      IF (pulse = '1') THEN --if key pressed check table to BCD
        IF (l_temp(0) = '1' AND C(0) = '1') THEN
          SIG <= "0001";
        ELSIF (l_temp(1) = '1' AND C(0) = '1') THEN
          SIG <= "0100";
        ELSIF (l_temp(2) = '1' AND C(0) = '1') THEN
          SIG <= "0111";
        ELSIF (l_temp(3) = '1' AND C(0) = '1') THEN
          SIG <= "1100";
        ELSIF (l_temp(0) = '1' AND C(1) = '1') THEN
          SIG <= "0010";
        ELSIF (l_temp(1) = '1' AND C(1) = '1') THEN
          SIG <= "0101";
        ELSIF (l_temp(2) = '1' AND C(1) = '1') THEN
```

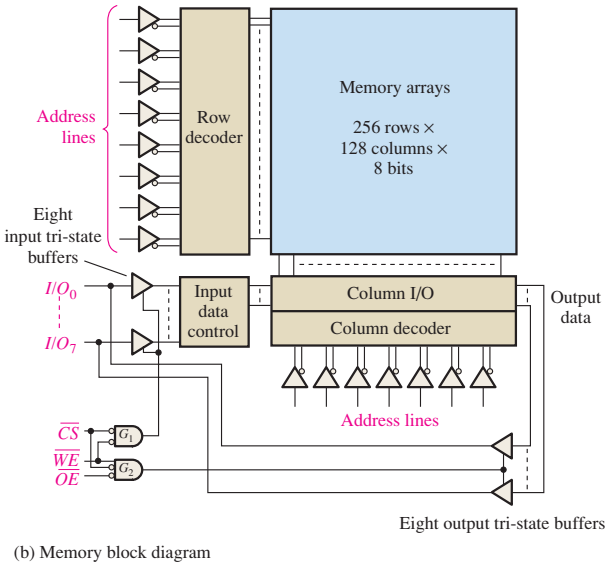
```
          SIG <= "1000";
        ELSIF (l_temp(3) = '1' AND C(1) = '1') THEN
          SIG <= "0000";
        ELSIF (l_temp(0) = '1' AND C(2) = '1') THEN
          SIG <= "0011";
        ELSIF (l_temp(1) = '1' AND C(2) = '1') THEN
          SIG <= "0110";
        ELSIF (l_temp(2) = '1' AND C(2) = '1') THEN
          SIG <= "1001";
        ELSIF (l_temp(3) = '1' AND C(2) = '1') THEN
          SIG <= "1111";
        ELSE
          SIG <= "1111";
        END IF;
      END IF;
    END IF;
  END PROCESS;
  L <= l_temp;
END ARCH;
```

2.2 Memory device

The auxiliary memory of a system has the duty of keeping essential information that the Central Processing Unit (CPU), in our case a GAL, will need in order to operate.

This information is usually kept in words of two bytes (16 bits in total), treated as individual entities or memory locations.

In order to increment the amount of data that can be stores, memory devices combine multiple memory locations into two dimensional arrays (i.e. 8x8 array), or even three dimensional arrays (i.e. 8x8x8 array). We can see this here:



(b) Memory block diagram

Figure 4: RAM block diagram [2]

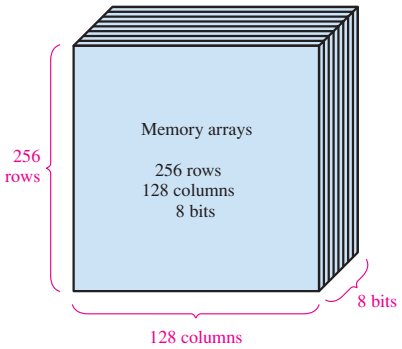


Figure 5: RAM array configuration [2]

2.2.1 Chip description

Chip Enable

To operate the memory first we have to activate the Chip Enable $\overline{CE}$ terminal (Active Low).

When a CPU has to handle multiple memory chips, in order to read or write from one of them, the other ones are disabled sending a HIGH to their respective $\overline{CE}$ terminals.

Write Enable

With this terminal the CPU specifies if the operation it wants to execute is to write into the memory (LOW at $\overline{WE}$) or to read from the memory (HIGH at $\overline{WE}$).

Output enable

The $\overline{OE}$ connects the output of the memory to the data bus. If it is in LOW, data can be sent from the memory. Otherwise, a HIGH level will create a High-Z output at the Data Bus. We usually keep it in a constant LOW in order to simplify the system.

Address bus

In order to read or write, the GAL must first specify which memory location it wants to work with. This is done by means of the address bus, corresponding to the row and the column of the memory matrix.

Mode	$\overline{\textbf{CS}}$	$\overline{\textbf{OE}}$	$\overline{\textbf{WE}}$	**I/O**
Standby	H	X	X	High-Z
Read	L	L	H	DATA_{OUT}
Read	L	H	H	High-Z
Write	L	X	L	DATA_{IN}

Table 2: Operation Modes Truth Table [3]

The RAM device used in the project is the 6116, a 16K (2048 x 8 bit) high speed memory chip.

In order to simplify the operation, as only 4 digits are stored, we will only use the A0 and A1 Address terminals.

Also, as we only need to store digits from 0 to 9, only the first 4 bits of the Data Bus will be used.

2.3 Safecode

2.3.1 Writing Operation

In order to be able to check the introduced password, we will program it into a Random Access Memory (RAM), in particular a 6116 CMOS Static one so as to be able to read it later on. To do this, we will use a GAL. In this case, the clock (CLK) and the BCD signal from the KEYPAD GAL (SIG) will be its inputs. In order for the RAM to work properly, it requires the SAFECODE GAL to have as output a Chip Select signal (CS__), an Address signal (A), and the BCD (D__).

As the programmable chip acts as a FSM, it will also have as output a representation of its current state, which will be later used to command other GAL devices. It should be noted that due to the fact that the RAM IC will be commanded by different GALs, we have added resistors between the output of the GAL and the control pins of the RAM so as to avoid short circuits. We can clearly see this in the following picture:

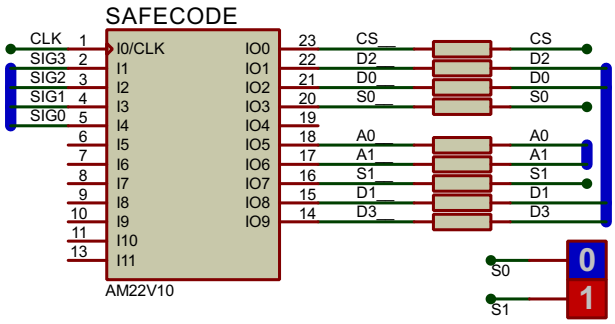


Figure 6: Proteus Subassembly of Safecode

The memory device will start storing data when the Write Enable ($\overline{WE}$) signal and the Chip Select ($\overline{CS}$) are pulled LOW by the GAL. A timing diagram is attached:

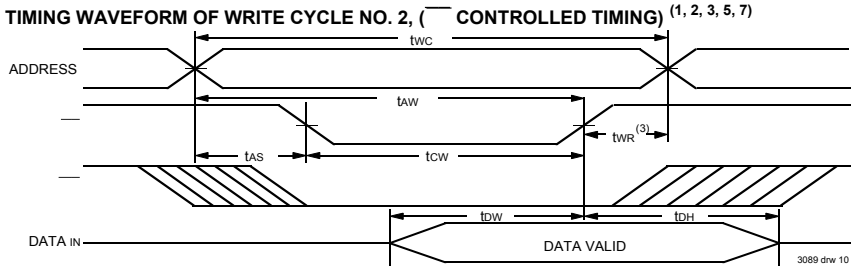


Figure 7: 6116 RAM Write Cycle

The general truth table of this specific RAM, though it is mostly the same for most RAM chips, can be found in **Table 2**

Now we will discuss the different states that we have defined in our FSM. As per before, the full code will be attached as well.

2.3.2 FSM States Overview

Q_0

At rest, the system must start at Address "00", and the $\overline{CS}$ signal must be HIGH because it is active when it is LOW. Once the $*$ button is pressed, system transitions to Q_1 state.

Q_1

The $\overline{CS}$ signal changes to a LOW level, and the RAM device is in the "ready to write" State Q_1. This means that from now on, Address number 00 is ready to be overwritten by the character we introduce with the keypad.

When introducing a new character with the keypad, the $*$ will make the system stay at Q_1, the $\#$ character will make the system to return to Q_0, and any other character will make the system change to State Q_2.

Q_2 & Q_3 *and return to* Q_0

In state Q_2, the character previously introduced is written on the address 00.

Now, there are two options: to keep writing on the memory, or to stop writing.

- In order to keep writing, any key pressed but $\#$ and the same key, will transition the system to state Q_3, where the system jumps to the next Address (where the next character will be kept), and the writing loop starts again at Q_1.
- In order to stop, the $\#$ button is pressed, and the system returns to state Q_0, meaning that the Address is reset to 00, and the writing operation ends.

To illustrate the different states and the transitions between them we have included the following diagram:

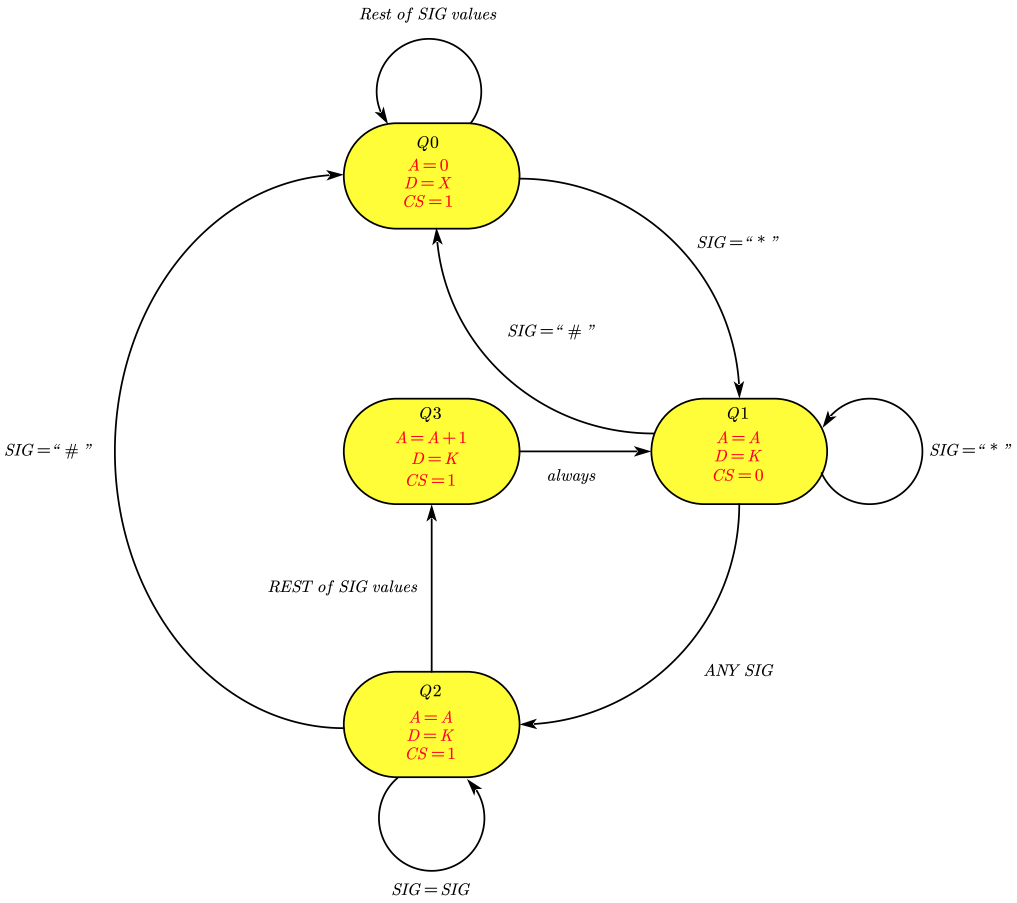


Figure 8: Safecode FSM

The VHDL code for this GAL can be found below:

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE IEEE.STD_LOGIC_ARITH.ALL;
4  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  ENTITY SAFECODE IS
7    PORT (
8      CLK : IN std_logic;
9      SIG : IN std_logic_vector(3 DOWNTO 0); --Keypad input
10     A : INOUT INTEGER RANGE 0 TO 3; --Address input & output
11     D : OUT std_logic_vector(3 DOWNTO 0); --Password output to RAM
12     CS : OUT std_logic --Chip select output
13   );
```

```
END SAFECODE;

ARCHITECTURE ARCH OF SAFECODE IS
  TYPE STATE_TYPE IS (Q0, Q1, Q2, Q3); --State declaration
  SIGNAL STATE : STATE_TYPE;
BEGIN
  PROCESS (CLK)
  BEGIN
    IF (rising_edge(CLK)) THEN
      CASE STATE IS --State 0, A=0, CS=1
        WHEN Q0 => A <= 0;
          CS <= '1';
          IF (SIG = "1100") THEN --If “*”
            STATE <= Q1; --Go to State 1
          ELSE
            STATE <= Q0; --Stay in State 0
          END IF;
        WHEN Q1 => D <= SIG; --State 1, D=SIG, A=A, CS=0
          CS <= '0';
          A <= A;
          IF (SIG = "1100") --If “*”
            STATE <= Q1; --Stay in State 1
          ELSIF (SIG = "1111") THEN --If “#”
            STATE <= Q0; --Go back to State 0
          ELSE --If not “#”
            STATE <= Q2; --Go to State 2
          END IF;
        WHEN Q2 => CS <= '1'; --State 1, D=SIG, A=A, CS=1
          IF (D = SIG) THEN --If D=SIG
            STATE <= Q2; --Stay in State 2
          ELSIF (SIG = "1111") THEN --If “#”
            STATE <= Q0; --Go to State 0
          ELSE
            STATE <= Q3; --Go to State 3
          END IF;
        WHEN Q3 => A <= A + 1; --State 3, D=SIG, A=A+1, CS=1
          STATE <= Q1; --Go to State 1
      END CASE;
    END IF;
  END PROCESS;
END ARCH;
```

2.4 State_Fsm & Display

Once the password has been entered, it is displayed in the 4 7-segment displays. This display must show the entered four digit password and the “Open” or “Error” messages, depending on whether code is correct or not. Since all of the checking/validating code cannot fit in one GAL. the display must be able to be controlled by different GAL devices.

In order to display the entered password, two different GALs, State_FSM and Display, will be used due to the length of the code. We will go over the "Open"/"Error" messages later.

2.4.1 State_Fsm

STATE_FSM has as input the clock (CLK), the $RESET$ signal, the output FSM states from the SAFECODE (S) as inputs as well (As we discussed in **Subsubsection 2.3.1**), and the $DONE$ signal as output.

Most of the GALs of this project use FSMs to command the different operations, including this one. We will now go over the different states that form the State Machine:

Q_0

Initially, $DONE$ is in a LOW level, then, the Final State Machine begins. The first state, Q_0, pulls $DONE$ LOW, and changes to the next state if the read state from the safecode $S = 00$.

Q_1, Q_2 and return to Q_0

At Q_1, DONE remains the same, and once $S = 10$, that is, the second state of the SAFECODE, DONE is pulled HIGH and the state changes to Q_2. No changes occur in Q_2 until RESET is in HIGH level, then the state becomes Q_0, effectively resetting the GAL.

To illustrate the different states and the transitions between them we have included the following diagram:

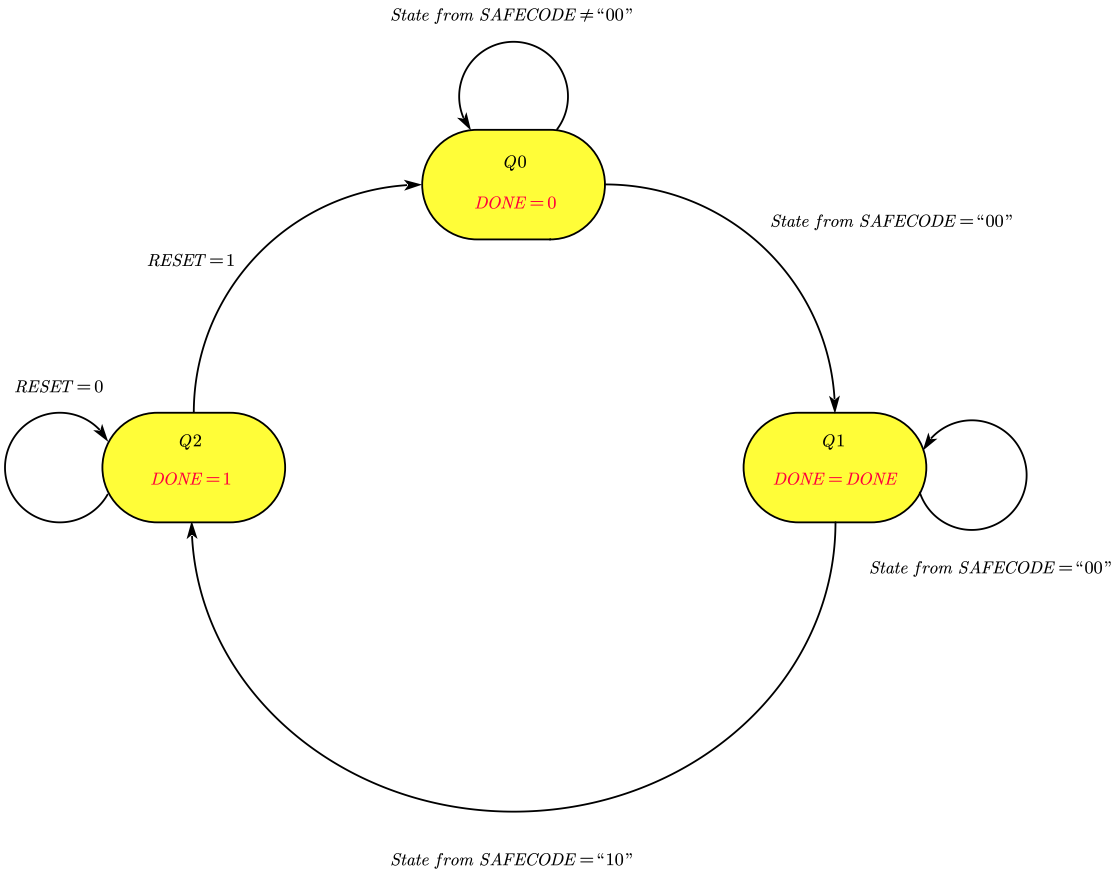


Figure 9: State_FSM FSM

The VHDL code for this GAL can be found below:

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY FSM IS
  PORT (
    CLK : IN std_logic;
    RESET : IN std_logic; --RESET input
    S : IN std_logic_vector(1 DOWNTO 0); --States from SAFECODE
    DONE : OUT std_logic := '0'; --DONE output
  );
END FSM;
```

```
12
13 ARCHITECTURE ARCH OF FSM IS
14   TYPE STATE_TYPE IS (Q0, Q1, Q2); --State declaration
15   SIGNAL STATE : STATE_TYPE;
16 BEGIN
17   PROCESS (CLK, RESET)
18   BEGIN
19     IF (rising_edge(CLK)) THEN
20       DONE <= '0';-- Initialize DONE=0
21       CASE STATE IS
22         WHEN Q0 => DONE <= '0'; --State 0, DONE=0
23           IF (S = "00") THEN --If State of SAFECODE 0
24             STATE <= Q1; --Go to State 1
25           END IF;
26         WHEN Q1 => DONE <= DONE; --State 1, DONE=DONE
27           IF (S = "10") THEN --If State of SAFECODE 2
28             DONE <= '1'; --DONE=1
29             STATE <= Q2; --Go to State 2
30           END IF;
31         WHEN Q2 => DONE <= DONE; --State 2, DONE=DONE
32           IF (RESET = '1') THEN --If RESET=1
33             STATE <= Q0; --Go back to State 0
34           END IF;
35       END CASE;
36     END IF;
37   END PROCESS;
38 END ARCH;
```

2.4.2 Display

DISPLAY has, as an input, the clock signal (CLK) and $DONE$ signal from STATE_FSM. Also, it has as outputs SEL, a 4 bit signal which acts as a ring counter for the 4 digit 7 segment displays, A_OUT, a 2-bit vector that controls the addressing of the RAM, WE_OUT and CS_OUT, being the Write Enable and Chip Select commanding signals for the RAM module.

Due to the fact that there is more than one GAL connected to the RAM to both read and write to it, as well as to the display, the use of tri-state buffers is necessary so as to avoid short circuits. The tri-state buffers act as switches commanded by CS_OUT, A_OUT & WE_OUT. The output of these "switches" are the $\overline{CS}$, A & $\overline{WE}$ signals needed to control the RAM IC.

We will now go over the different states that form this Finite State Machine:

Initialization Stage

Every time that DONE is in LOW level, all the outputs are initialized at '0'.

Ring (Q_0, Q_1, Q_2, Q_3)

Once DONE changes its value to HIGH, WE_OUT and CS_OUT switch to HIGH level as well, enabling the tri-state buffers and pulling $\overline{WE} = 1$ and $\overline{CS} = 0$ in order to put the RAM into Read Mode so as to show, at [D0...D3] the corresponding number depending on the address A_OUT.

So, in the FSM of this GAL, in each of the states, A_OUT will have a value for an address, at the same time, SEL will change the position of its HIGH level bit, turning one of the 4 7-segment display ON at a time, with the rest being OFF. Since the clock frequency is high, displaying only one number per clock frequency is enough to trick the human eye into believing that the 4 numbers are being displayed at the same time. This is done in such a way that the sequence of numbers that are being shown correspond to the ones stored in the RAM.

End of Ring

If the ring passes through Q_0 and the DONE signal is in a LOW level, the ring stops and the system returns to the initialization stage.

To show the numbers in the display, we have fashioned a custom 7-segment display driver. We can see it in the following image:

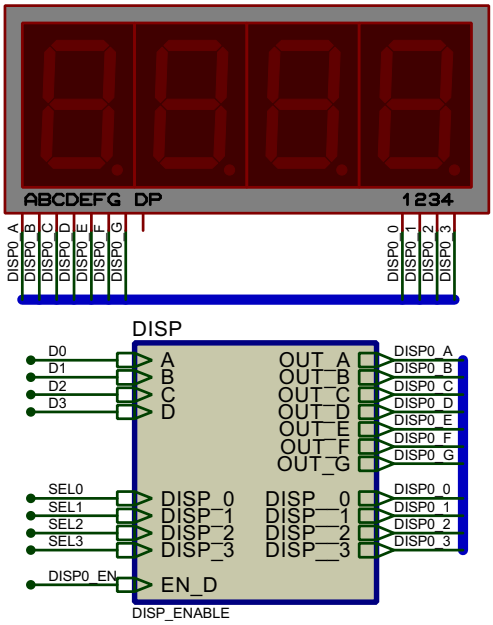


Figure 10: Proteus Subassembly of 7-segment display driver

The internals of this driver are composed of a 74247 display driver and some tri-state buffers to control when the driver is on, as well as to prevent short circuits in the outputs of the driver/inputs of the display. As we said before, this is due to the fact that the display is driven by different GALs, so we want a High-Z output in the drivers that are not in use. The EN_D signal is provided by another GAL that takes care of selecting which signal to display. (See **Subsubsection 2.10.1** for more on this). We can see this internals here:

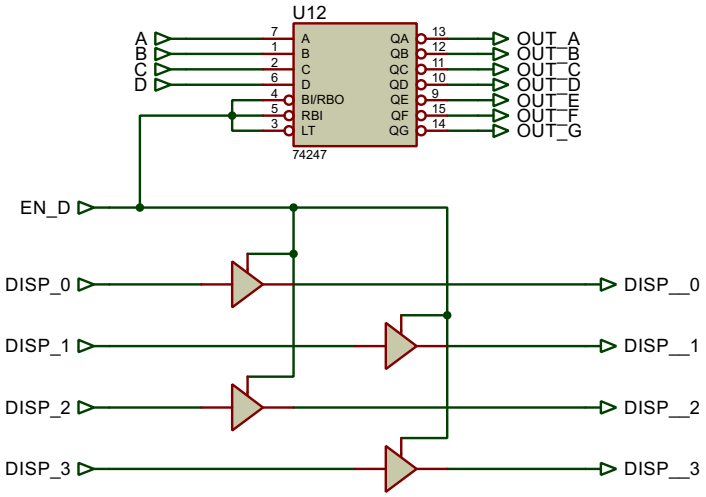


Figure 11: 7-segment display driver internals

To illustrate the different states and the transitions between them we have included the following diagram:

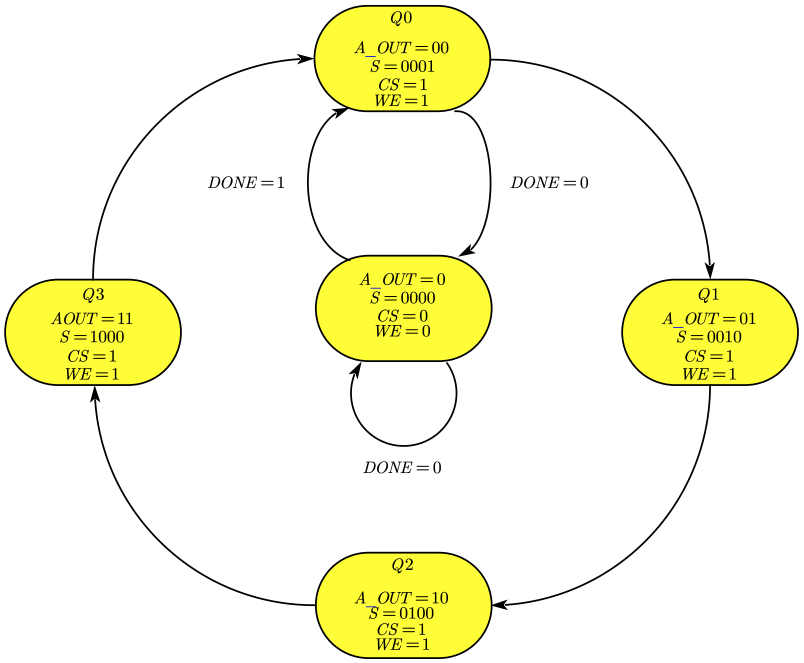


Figure 12: Display FSM

The final subassembly of both GALs can be seen here:

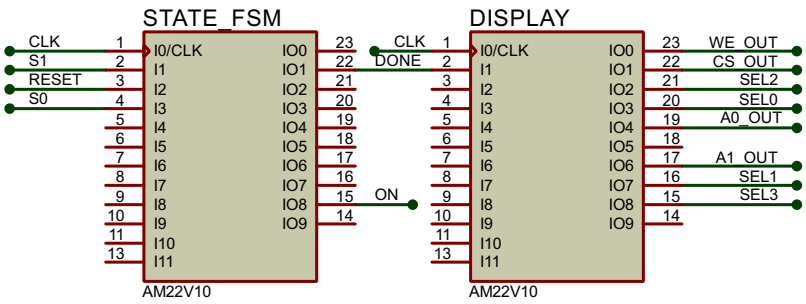


Figure 13: Proteus Subassembly of both GALs

The VHDL code is attached below:

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY DISPLAY IS
  PORT(CLK : IN std_logic;
       DONE : IN std_logic; --DONE input
       SEL : OUT std_logic_vector(3 DOWNTO 0); --DISP select output
       A_OUT : OUT std_logic_vector(1 DOWNTO 0); --ADDR enable output
       WE_OUT : OUT std_logic; --Write enable output
       CS_OUT : OUT std_logic --Chip select output
END DISPLAY;

ARCHITECTURE ARCH OF DISPLAY IS
   TYPE STATE_TYPE IS (Q0, Q1, Q2, Q3); --State declaration
   SIGNAL STATE : STATE_TYPE;
BEGIN
   PROCESS (CLK)
      BEGIN
         IF (rising_edge(CLK)) THEN
            IF (DONE = '0') THEN --If DONE=0
               CS_OUT <= '0'; --Chip select=0
               WE_OUT <= '0'; --Write enable=0
               SEL <= "0000"; --DISPLAY select output=0
               A_OUT <= "00"; --ADDRESS enable output=0
            ELSE --If DONE=1
               CS_OUT <= '1'; --Chip select=1
               WE_OUT <= '1'; --Write enable=1
               CASE STATE IS
                  WHEN Q0 => A_OUT <= "00"; --State 0, A_OUT=0, SEL=1
                             SEL <= "0001";
                             STATE <= Q1; --Go to State 1
                  WHEN Q1 => A_OUT <= "01"; --State 1, A_OUT=1, SEL=2
                             SEL <= "0010";
                             STATE <= Q2; --Go to State 2
                  WHEN Q2 => A_OUT <= "10"; --State 2, A_OUT=3, SEL=3
                             SEL <= "0100";
                             STATE <= Q3; --Go to state 3
                  WHEN Q3 => A_OUT <= "11"; --State 3, A_OUT=4, SEL=4
                             SEL <= "1000";
                             STATE <= Q0; --Go back to State 0
               END CASE;
            END IF;
         END IF;
   END PROCESS;
END ARCH;
```

2.4.3 Final Roundup

We know that this section may strike as a bit complex or hard to understand, so we will try to break down the different actions that both GALs perform in an step by step description of the procedure.

1. The STATE_FSM reads the state of SAFECODE using the state outputs of the latter.

2. Once it detects a "00" it knows that the password is being entered.

3. As soon as it reads a "10" (Last State of SAFECODE, triggered by the # key), it knows that all numbers have been entered and so it turns the *DONE* pin ON.

4. DISPLAY reads the *DONE* pin and as soon as it turns ON, it starts a ring counter that sweeps through the 4 7-segment displays activating one each time. At the same time, the RAM is put into Read mode and the addresses are swept as well. In essence, the first displays turns on (with the rest off) and it displays the contents of Address = "00", then the second display turns on and displays Adress "01", and so on. This happens very fast, tricking the person into believing that all 4 displays are ON simultaneously.

5. Once the *RESET* signal is received by STATE_FSM, the *DONE* pin is pulled LOW and both GALs are reset.

2.5 Digit 0, 1, 2, 3

Once the 4 digit password is entered with the keypad, the system has to compare it with the password preset by the user with the dip-switches.

In order to do this, each of the 4 digits of the password is compared by a different GAL due to a limitation in the number of input pins. This example corresponds to the DIGIT_0 of the password, but the other three work in the same way, thus, only one of them is explained.

For the GAL to know which of the digits is the first one, the Address (A) from the memory device is read. In this case, if the Address is 00, we have the digit required at the Data Bus (D).

We will now go over the different states of the Finite State Machine:

Q_0

By default, the Output of the GAL, *DIGIT_0* is at a LOW state. The GAL checks 4 inputs, namely *DONE*, the predefined password (via the dip switch) *PASS0*, the address value, *A* and the value stored in the RAM at that specific address, *D*. If the *DONE* pin is ON, meaning that the whole password has been introduced, the address is "00" (for this particular digit), and *PASS0* = *D*, then the state is changed and the FSM moves to *Q1*.

Q_1

In this state, the GAL device will output a HIGH at *DIGIT_0* indicating that the first digit of the password is correct.

Finally, when the *RESET* signal is detected, the state switches back to Q0 and the *DIGIT_0* pin is pulled LOW, effectively resetting the FSM.

To illustrate the different states and the transitions between them we have included the following diagram:

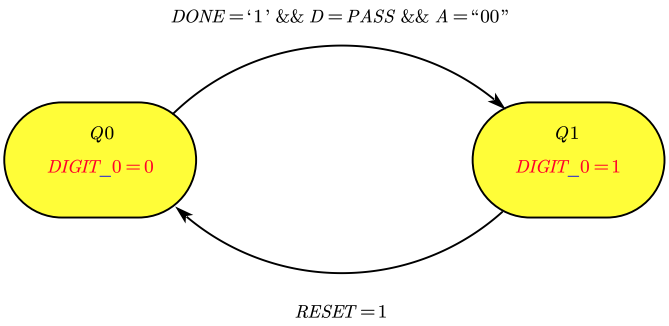


Figure 14: Digits [0...3] FSM

The VHDL code describing this subsystem is attached below:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY DIGIT0 IS
  PORT(CLK : IN std_logic; --CLOCK input
       RESET : IN std_logic; --RESET input
       A : IN std_logic_vector(1 DOWNTO 0); --ADDRESS input
       D : IN std_logic_vector(3 DOWNTO 0); --password in keyboard
       PASS0 : IN std_logic_vector(3 DOWNTO 0); --correct password
       DONE : IN std_logic; --Done input
       DIGIT_0 : OUT std_logic --"correct or incorrect digit" out);
END DIGIT0;

ARCHITECTURE ARCH OF DIGIT0 IS
  TYPE STATE_TYPE IS (Q0, Q1); --State declaration
  SIGNAL STATE : STATE_TYPE;
BEGIN
  PROCESS (CLK, RESET)
  BEGIN
    IF (rising_edge(CLK)) THEN
      DIGIT_0 <= '0'; --initialize OKAY=0
      CASE STATE IS
        WHEN Q0 => DIGIT_0 <= '0'; --State 0, OKAY=0
          IF (DONE = '1' AND D = PASS0 AND A = "00") THEN
            --If DONE && password = correct password && A=00
            STATE <= Q1; --Go to State 1
          END IF;
        WHEN Q1 => DIGIT_0 <= '1';--State 1, OKAY=1
          IF (RESET = '1') THEN --If RESET=1
            STATE <= Q0; --Go back to State 0
          END IF;
      END CASE;
    END IF;
  END PROCESS;
END ARCH;
```

The Proteus subassembly corresponding to the 4 digits can be seen below:

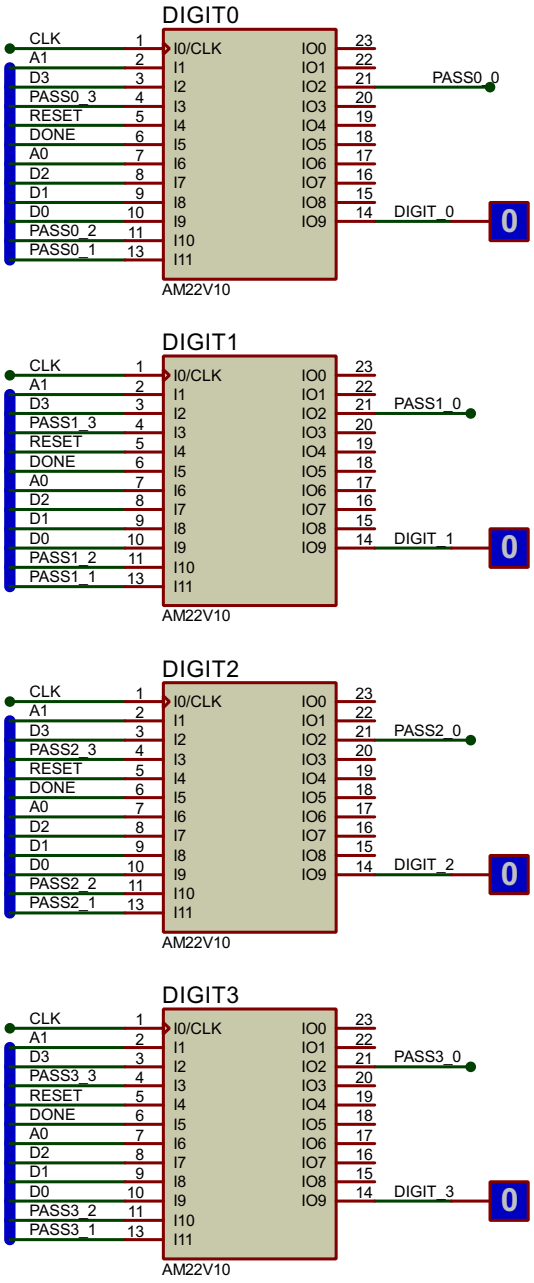


Figure 15: Proteus Subassembly of Digits [0...3]

2.6 Password checking

Once every digit has been checked individually, all of them have to be checked to determine whether the whole password is correct or not. To do this, we will make use of another GAL, namely *PASSWORD CHECK*.

The latter has as an input, the clock (*CLK*), *DONE_*, which is the delayed version of DONE (See **Subsection 2.8.1**), the 4 *DIGIT_* signals coming from the individual checks and the general *RESET*. As outputs, it has *CORRECT* signal, and *SENT_PULSE_*, a signal that is used to initialize the time-sensitive operations which purpose is to display the introduced password in the 4 7-segment displays followed by the ***Open/Error*** message and finally, once everything has been displayed, to reset the whole circuit.

The *DONE_* signal is a slighty delayed version of the original *DONE* signal. This is needed because otherwise the individual digit signals, which are normally LOW, arrive **AFTER** the *DONE* one, creating a point in time in which the *DONE* is **HIGH**, which would indicate that the password has been entered, but the *DIGIT_* signals are still **LOW** since the GALs in **Subsection 2.5** have not had the time to process the digits yet. Delaying the *DONE* signal by some ms, a time higher than the propagation delay of the GALs, solves this problem.

We will now go over all the states that form the FSM:

Q_0

The code follows the operation of a FSM with 3 states. In Q_0, *CORRECT* and *SENT_PULSE* are initialized as LOW level, and remain in this condition until a HIGH is read in the delayed version of *DONE*, *DONE_*. Depending on the value of *DIGIT_* in that moment, the next state will either be Q_1 or Q_2.

If all bits of the *DIGIT_* array are on a HIGH level, it means that every digit stored in the RAM coincides in magnitude and order to the ones stored in the dip switches inside the security box. In this case the next state is Q_1.

On the other hand, if all bits of the *DIGIT_* array are NOT on a HIGH level, the state will change to Q_2.

Q_1

If the GAL is in this state, it will pull ***CORRECT*** and ***SENT_PULSE*** **HIGH** and it will wait for the *RESET* signal. Once received, the state will toggle to Q_0 effectively resetting the GAL.

Q_2

In this state the GAL will pull the ***CORRECT*** signal **LOW**, so as to indicate that it is NOT correct, and it will pull the ***SENT_PULSE*** **HIGH**, initializing the time-sensitive sequence that we discussed earlier. As per the last state, when a HIGH is received in the *RESET* pin, the state will toggle to Q_0 effectively resetting the GAL.

To illustrate the different states and the transitions between them we have included the following diagram:

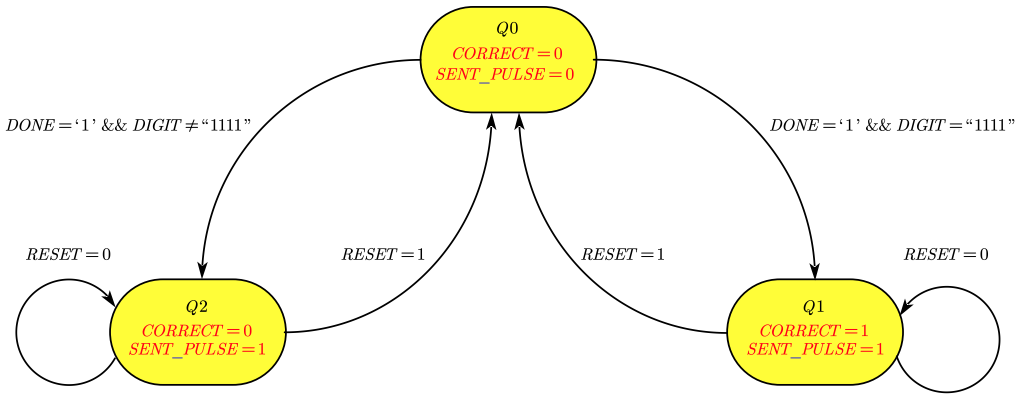


Figure 16: Password Check FSM

The VHDL code describing the operation of the FSM is attached below:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY MULTIPLIER IS
  PORT (
    CLK : IN std_logic;
    DONE_s : IN std_logic; --DONE_s input
    DIGIT_s : IN std_logic_vector(3 DOWNTO 0); --out from GAL DIGIT
    CORRECT : OUT std_logic; --password check
    SENT_PULSE : OUT std_logic; --exit pulse
    RESET : IN std_logic --RESET input
  );
END MULTIPLIER;

ARCHITECTURE ARCH OF MULTIPLIER IS
  TYPE STATE_TYPE IS (Q0, Q1, Q2); --State declaration
  SIGNAL STATE : STATE_TYPE;
BEGIN
  PROCESS (CLK)
  BEGIN
    IF (rising_edge(CLK)) THEN
      CASE STATE IS
        WHEN Q0 => CORRECT <= '0';
          SENT_PULSE <= '0';
          IF (DONE_s = '1' AND DIGIT_s = "1111") THEN
            STATE <= Q1; --when DONE_s and all correct change state
```

```
27            ELSIF (DONE_s = '1' AND DIGIT_s /= "1111") THEN
28              STATE <= Q2; --some DIGIT output is not one so INCORRECT
29            ELSE
30              STATE <= Q0;
31            END IF;
32          WHEN Q1 => CORRECT <= '1';
33            SENT_PULSE <= '1';
34            IF (RESET = '1') THEN
35              STATE <= Q0; --if RESET then restart
36            ELSE
37              STATE <= Q1; --if not, stay in Q1
38            END IF;
39          WHEN Q2 => CORRECT <= '0'; --INCORRECT= !CORRECT
40            SENT_PULSE <= '1';
41            IF (RESET = '1') THEN
42              STATE <= Q0; --if RESET then restart
43            ELSE
44              STATE <= Q2; --if not, stay in Q2
45            END IF;
46        END CASE;
47      END IF;
48    END PROCESS;
49  END ARCH;
```

Finally, the Proteus subassembly of the circuit can be found below:

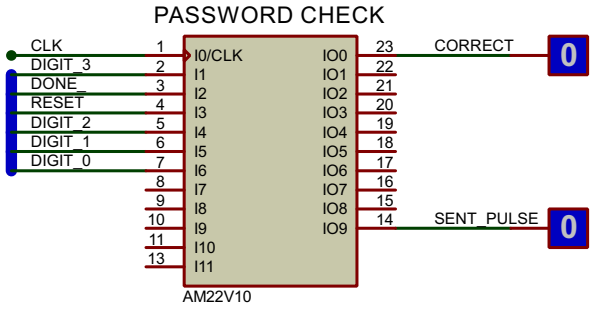


Figure 17: Proteus Subassembly of Password Check

2.7 Open/Error Message Displaying

Once the system has determined whether the password entered via the keypad is correct or not, an HMI (Human-Machine Interface) must be implemented.

In order to do this, we will display the result of the checking process in the 4 7-segment displays that we have already seen. 2 GALs, in combination with some special circuitry that we have fashioned ourselves will be in charge of displaying the messages: Either ***Open***, when the code entered is the correct one, or ***Err*** when it is not.

As we have said before, each message will be processed by one GAL device due to a constraint in the number of pins. We will now go over the code of both GALs and the custom circuitry that allows us to display the messages:

2.7.1 *Open* Message

As per most of the GALs that we have used, both of them use a FSM to command the different operations. Since the codes of both are almost equal, the I/O declaration of pins will be the same.

On the one hand, as an input, we may find the signals clock (*CLK*), the *DONE* signal (Not the delayed version this time), the *CORRECT* signal, which we thoroughly discussed in **Subsection 2.6**, the *INCOMING_PULSE* signal, which is a delayed pulse of the signal *SENT_PULSE* that again, we discussed in **Subsection 2.6** and finally, the *RESET* signal, which will reset the FSM back to the original state.

On the other hand, as an output, we may find the signals *NUMBER* which control the message displaying process, the *DISP_EN*, which, as the name suggests, enables the display and, finally, *ENDPULSE_SEND* which resets the whole system after a few ms, so as to allow the user to visualize the Open/Error message for a brief amount of time (See **Subsection 2.8.2** for more on this). Otherwise, the whole Opening/Error sequence would happen too fast for the user to notice.

As we have said, the 2 GALs have almost identical codes, that is why we are only going to describe their FSM once. For instance the *OPEN* GAL's FSM can be broken down into the following states:

Q_0

The system initializes with *ENABLE* and *ENDPULSE_SEND_O* in a LOW level. Only if *DONE*, the *CORRECT* and the *INCOMING_PULSE* pins are in a HIGH state, the system will change to the next state, Q_1.

Q_1, Q_2, Q_3 & Q_4

When the system enters in the Q_1 state, a ring counter starts. As per the DISPLAY GAL, this counter sweeps across the 4 7-segment displays activating only one at the time at very high frequency, enough to make the illusion that the four displays are ON at the same time.

The only issue that we ran into was displaying the words *Open*/*Err*, due to the pin limitations of the GALs and their small capacity. To solve this, we created our own drivers using a combination of tri-state buffers to provide power to the needed segments out of the 7 available for each particular letter. We can see this here:

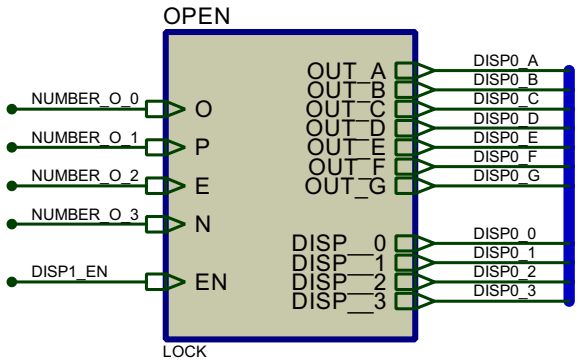


Figure 18: Custom Fixed Open Message Driver

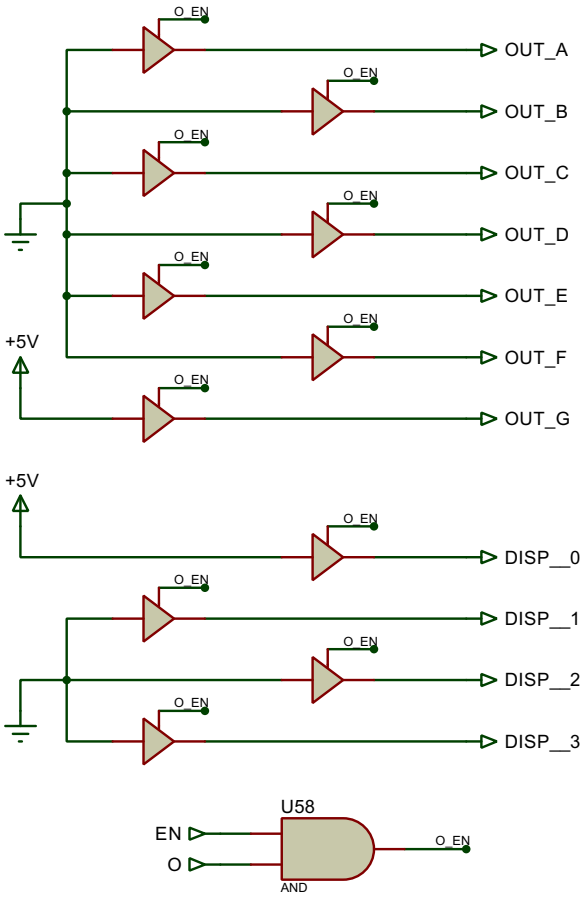


Figure 19: Driver internals (Letter "O")

These drivers take the *NUMBER* signal as well as the ENABLE SELECTOR's *ENABLE* one as inputs. When the GALs starts the ring, the *ENABLE* signal is pulled HIGH, and based on the choice of the *ENABLE SELECTOR* GAL (See **Subsection 2.10** for more on this topic) the driver outputs both the display address, i.e., which 7-segment display needs to be turned on, as well as the state of the 7 segments. This allows us to create custom messages using a reduced amount of pins.

Of course, this method is rudimentary and not very efficient nor cost effective, but it was necessary for the correct visualization of the data.

To conclude, each time that the ring passes through $\boldsymbol{Q_4}$, i.e., the state corresponding to letter *N*, the system checks the level of the *RESET* signal. If it is in a HIGH level, the ring finishes and returns to state Q_0. If it is on a LOW level, the loop continues. This effectively resets the GAL once the message displaying procedure is finished.

To illustrate the different states and the transitions between them we have included the following diagram:

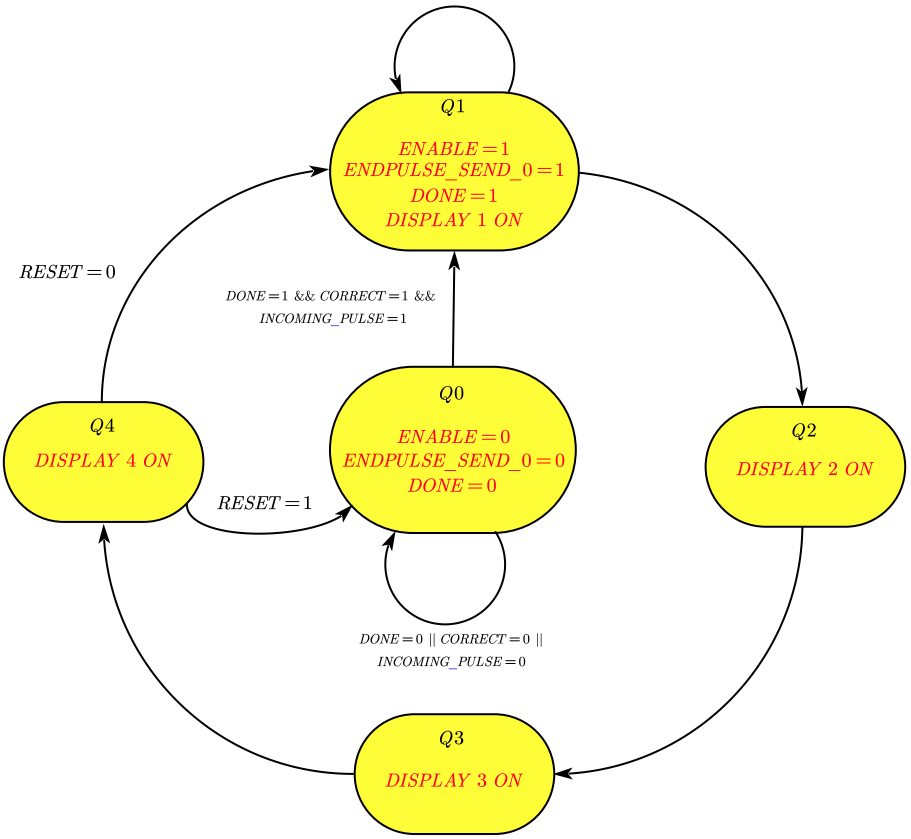


Figure 20: Open FSM

The VHDL code that describes the process can be seen below:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY OPEN_DOOR IS
  PORT (
    CLK : IN std_logic;
    INCOMING_PULSE : IN std_logic; --INCOMING_PULSE input
    DONE : IN std_logic; --DONE input
    CORRECT : IN std_logic; --CORRECT PASSWORD input
    NUMBER : OUT std_logic_vector(0 TO 3); --Ring for display output
    ENABLE : OUT std_logic; --ENABLE output
    ENDPULSE_SEND_O : OUT std_logic; --ENDPULSE_SEND_O output
    RESET : IN std_logic --RESET input
  );
END OPEN_DOOR;

ARCHITECTURE ARCH OF OPEN_DOOR IS
  TYPE STATE_TYPE IS (Q0, Q1, Q2, Q3, Q4); --State declaration
  SIGNAL STATE : STATE_TYPE;

BEGIN
  PROCESS (CLK)
  BEGIN
    IF (rising_edge(CLK)) THEN
      CASE STATE IS

        WHEN Q0 => ENABLE <= '0'; --initialize ENABLE=0
          ENDPULSE_SEND_O <= '0'; --initialize ENDPULSE_SEND=0

          IF (DONE = '1' AND CORRECT = '1'
          AND INCOMING_PULSE = '1') THEN
          --If Done=1 && Correct=0 && Incoming_pulse=1
            STATE <= Q1; --Go to state 1
          ELSE
            STATE <= Q0; --Stay at state 0
          END IF;
          -- RING INITIALIZE
        WHEN Q1 => ENABLE <= '1'; --ENABLE=1
          ENDPULSE_SEND_O <= '1'; --ENDPULSE_SEND=1
          NUMBER <= "1000"; --DISPLAY 1 ON
          STATE <= Q2; --Go to state 2

        WHEN Q2 => NUMBER <= "0100"; --DISPLAY 2 ON
          STATE <= Q3;--Go to state

        WHEN Q3 => NUMBER <= "0010"; --DISPLAY 3 ON
```

```
46            STATE <= Q4;--Go to state 4
47
48          WHEN Q4 => NUMBER <= "0001"; --DISPLAY 4 ON
49
50            IF (RESET = '1') THEN --If RESET=1
51              STATE <= Q0; --Go back to state 0 (FINISH DISPLAYING)
52            ELSE
53              STATE <= Q1; --Go back to state 1 (KEEP DISPLAYING)
54            END IF;
55        END CASE;
56      END IF;
57    END PROCESS;
58  END ARCH;
```

2.7.2 *Error* Message

The ***Error*** message works in the same way as the ***Open*** message, though there are two main differences:

- The *CORRECT* signal must be in a LOW level as we have to display the error message when the password entered by the user is incorrect.
- As the ***Err*** message has three characters and not four, the fourth display and therefore the last state Q_4 are not used.

To illustrate the different states and the transitions between them we have included the following diagram:

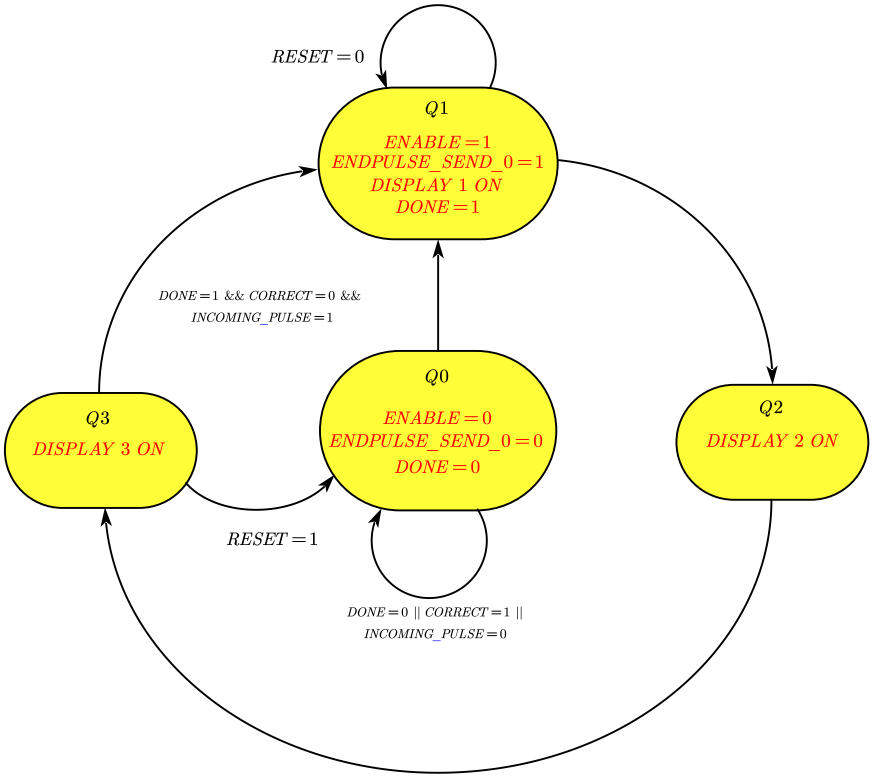


Figure 21: Error FSM

The VHDL code that describes the process can be seen below:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY ERROR_DOOR IS
  PORT (
    CLK : IN std_logic;
    INCOMING_PULSE : IN std_logic;-- INCOMING_PULSE input
    DONE : IN std_logic; --DONE input
    CORRECT : IN std_logic; --CORRECT PASSWORD input
    NUMBER : OUT std_logic_vector(0 TO 2); --Ring for display output
    ENABLE : OUT std_logic; --ENABLE output
    ENDPULSE_SEND_E : OUT std_logic; --ENDPULSE_SEND_E output
    RESET : IN std_logic --RESET input
  );
END ERROR_DOOR;

ARCHITECTURE ARCH OF ERROR_DOOR IS
  TYPE STATE_TYPE IS (Q0, Q1, Q2, Q3); --State declaration
  SIGNAL STATE : STATE_TYPE;

BEGIN
  PROCESS (CLK)
  BEGIN
    IF (rising_edge(CLK)) THEN
      CASE STATE IS

        WHEN Q0 => ENABLE <= '0'; --initialize ENABLE=0
          ENDPULSE_SEND_E <= '0'; --initialize ENDPULSE_SEND=0

          IF (DONE = '1' AND CORRECT = '0'
          AND INCOMING_PULSE = '1') THEN
          --If Done=1 && Correct=0 && Incoming_pulse=1
            STATE <= Q1; --Go to state 1
          ELSE
            STATE <= Q0; --Stay at state 0
          END IF;
          -- RING INITIALIZE
        WHEN Q1 => ENABLE <= '1'; --ENABLE=1
          ENDPULSE_SEND_E <= '1'; --ENDPULSE_SEND=1
          NUMBER <= "100"; --DISPLAY 1 ON
          STATE <= Q2; --Go to state 2

        WHEN Q2 => NUMBER <= "010"; --DISPLAY 2 ON
          STATE <= Q3;--Go to state 2

        WHEN Q3 => NUMBER <= "001"; --DISPLAY 3 ON
```

```
46
47            IF (RESET = '1') THEN --If RESET=1
48              STATE <= Q0; --Go back to state 0 (FINISH DISPLAYING)
49            ELSE
50              STATE <= Q1; --Go back to state 1 (KEEP DISPLAYING)
51            END IF;
52          END CASE;
53        END IF;
54      END PROCESS;
55    END ARCH;
```

The Proteus Subassembly of both circuits is attached below:

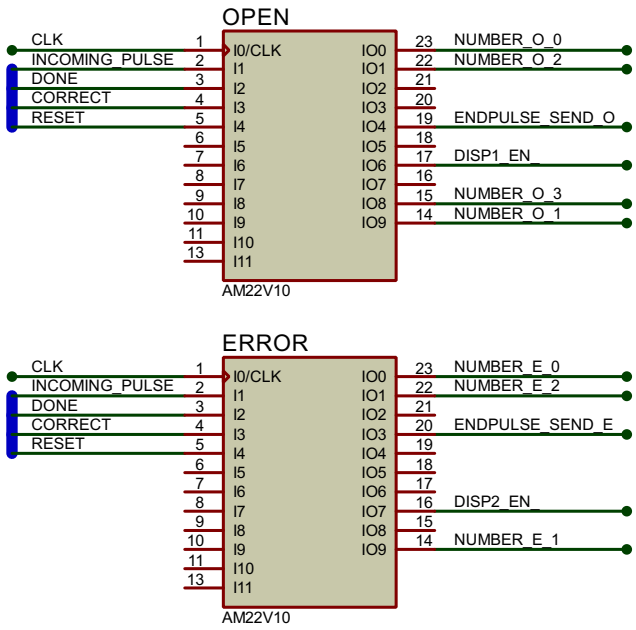


Figure 22: Open/Error Proteus Subassembly

The custom driver for the ***Err*** message, as well as its internals will not be shown as they are very similar to the one shown above.

2.7.3 Final Roundup

As per the last complex functional block, we will try to briefly go over the operation of the digit validation procedure since we know that it is a bit too hard to grasp. The different time-sensitive actions will be explained step by step.

1. The code is introduced by the user
2. Once the procedure described in **Subsubsection 2.4.3** is over (Storing the code in the RAM and displaying the introduced number), the *DONE* signal is pulled HIGH.
3. This signal is delayed approximately 0.5s to wait for the individual validation of the digits to occur (See **Subsection 2.5** for more on this topic).
4. The delayed version of the *DONE* signal arrives to the *Password Checking* GAL, as well as the results of the individual digit checking procedure. If the introduced combination is correct, this GAL pulls *CORRECT* HIGH, otherwise, it pulls it LOW. In either case, the GAL also pulls *SENT_PULSE* HIGH, which starts a 1 second delay in which the introduced digits are displayed.
5. After 1 second, the delayed signal reaches the OPEN/ERROR GALs. Based on the state of the *CORRECT* pin they decide whether to display the ***Open*** message or the ***Err*** one by pulling the corresponding *ENABLE* pin HIGH. This pin is read by the ENABLE SELECTOR GAL, that is there to avoid short-circuiting the displays (See **Subsection 2.10** for more on this topic). This last GAL turns on the correct driver, and the custom 7-segment display driver shows the appropriate message. At the same time, the *ENDPULSE_SEND* signal is pulled HIGH, activating yet another 1 second delay during which the *Open/Err* message is shown in the 4 7-segment displays.
6. After the 1 second delay is finished, the ***RESET*** signal is pulsed and the whole systems restarts, allowing the user to introduce another password (See **Subsection 2.9** for more on this).

2.8 Delays

Delays are one of the most crucial parts of this project since most of the system is time-sensitive, meaning that there are specific time intervals that nearly all programs have to take into account to function properly. Both monostable and astable timers can be implemented using clock cycle counters in VHDL, but for this project, we have decided to use the well-know 555 timer so as to reduce the amount of GALs. We can define 2 different 555 timer configurations in our project:

- Delayed steady ON after a PGT
- Delayed impulse after a PGT

None of the two configurations are standard, meaning that they can't simply be referred to as a barebones monostable or astable configuration. We have used non-linear elements such as diodes to achieve the desired results. We will now go over them so as to explain how they work and what their purpose is:

2.8.1 Delayed steady ON after a PGT

As the name suggests, this first configuration turns the output Q (Pin 3) to a high state one second after a PGT occurs in the input. In this case, the input is connected to the 555's V_{cc} pin. The circuit's diagram is as follows:

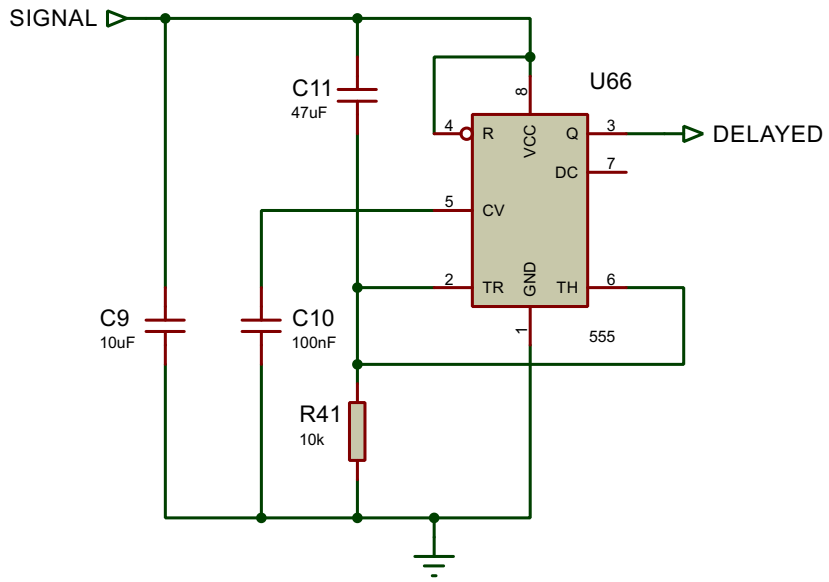


Figure 23: Delayed steady ON after a PGT

How this circuit works is through the RC network. The combination of the resistor and capacitor forms the RC network. This network determines the length of time it takes to charge the capacitor, i.e. the bigger the resistor and capacitor value, the longer the delay.

The reason why the circuit doesn't turn on automatically is because pin 2, the trigger pin, initially when the power turns on, is HIGH. This is because the capacitor hasn't charged up yet. Until the capacitor charges up, this pin is HIGH. Since the trigger pin is active LOW, the output will be off until this pin goes LOW. As the capacitor charges up and gets near the supply voltage it is connected to, the voltage at pin 2 decreases. When the voltage at pin 2 gets below 1/3 of the supply voltage, the pin is now LOW. When it is LOW, this is when the output goes HIGH, effectively delaying the input signal.

In this circuit, capacitors C9 and C10 are there just to decouple the input signal. They have nothing to do with the timing circuit.

This subassembly is used only once in our circuit. Its function is to delay the *DONE* signal. The purpose of this is to give the Individual Digit Checking GALs some time to perform their operations before checking if the whole password is correct. This delay is needed to overcome the internal propagation delays of the GALs.

2.8.2 Delayed impulse after a PGT

This second circuit is very similar to the first one, the only difference being that instead of a steady ON after a PGT, it only outputs a short pulse after a delay. To achieve this behaviour two 555s have been connected to one another. The first one is connected in the same configuration that we described earlier, that is, it outputs a delayed steady ON after a PGT. The second 555 circuit is a bit more complex. In essence, it behaves as a monostable, as it only outputs a single pulse when triggered. The only difference is that to trigger a monostable, a NGT pulse has to be applied to the Trigger pin (Pin 2), but this cannot be achieved using out configuration as the output of the first 555 is a steady ON.

To get around this problem, we have used a special tipe of monostable which operation we will describe below.

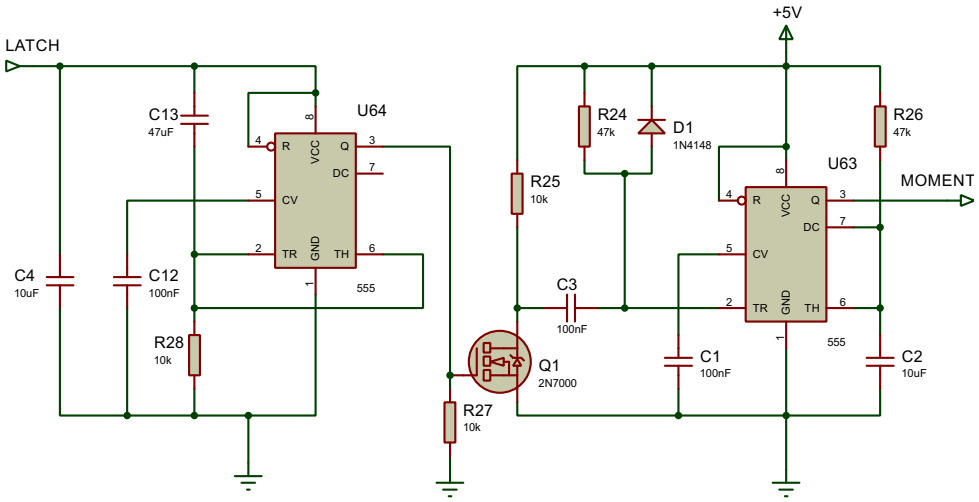


Figure 24: Delayed impulse after a PGT

When Q1 is off, that is, when the output of the first monostable is 0V, C3 is in short circuit, meaning that both its leads are at 5V due to the pull-ups R24 and R25, so the voltage drop across it is 0V. The voltage at the Trigger pin (Pin 2) is 5V as well, which means that the output Q (Pin 3) is off, as we need a voltage equal or lower than $V_{cc}/3$ at the Trigger pin to turn it on.

Once the N-Channel Mosfet is turned on by the first 555, the capacitor is grounded and an RC network is formed by R24 and C3. The voltage in the capacitor is 0V initially, so the voltage in the Trigger pin is 0V as well, turning the output on. As soon as C3 charges, The voltage at the trigger returns to 5V again and the output is switched off. In essence, this simulates the press of a button.

Q1 is an arbitrary N-Channel Mosfet and R27 is there to discharge its internal capacitor.

In our circuit, we have used this circuit three times to accomplish various duties such as:

- Auto starting the system. (See **Subsubsection 2.10.2**)
- Resetting the system once the execution of the code is finished. (**See Subsection 2.9**)
- To wait a few ms before showing the Open/Error message during which the introduced number is shown. (**See Subsections 2.6 and 2.7**)

2.9 General reset

Once the ***Open***/***Error*** message has been displayed for a few ms, all programmable logic devices must be reset.

In order to do this, the *RESET* signal that commands all the GALs, is operated with *ENDPULSE_SEND* output. This signal can come from two GALs, the *OPEN* and the *ERROR* one, and it indicates when the cycle is over, i.e. when the Open/Error message has been displayed for a few ms and the status LED has either turned red or green. To save some money and space, both of these signals have been connected to an OR gate which output is fed directly into one of the timers mentioned in **Subsubsection 2.8.2**. The timer's output is connected to the *RESET*, which resets all the state machines.

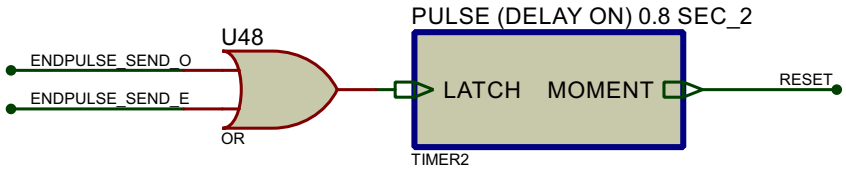


Figure 25: General Reset Proteus Subassembly

2.10 Enable Selector & Autostart

2.10.1 Enable Selector

Taking into account the fact that the 4-digit 7-segment displays receive data from different GALs throughout the different stages of the system, short-circuiting the data lines is very easy. We must therefore prevent this from happening so as to have a functioning system.

To solve this nuisance, an auxiliary GAL is needed. This device will command the *ENABLE* pins of out custom-designed drivers (See **Subsubsections 2.4.2 and 2.7.1** for more information on this topic). This, in essence avoids short circuits by only having one display driver active at the time (The rest being in High-Z mode).

The subsystem has the following inputs: the clock signal (*CLK*), the display signal (*DISP_EN_*), which comes from the OPEN/ERROR GALs, and the *READY* signal for the autostart.

On the other hand, the outputs are: *START*, which commands the beginning of the auto start, and the *DISP_EN* array that regulates the enable signals of the different sub-circuits.

As *DISP_EN_0* corresponds to the ***Open*** message, each time that it gets pulled HIGH, the *DISP_EN1*, which corresponds to the OPEN subcircuit enable signal, must be pulled HIGH too.

The same principle applies to *DISP_EN_1*. This signal corresponds to the ***Err*** message, so each time that it is pulled HIGH, the *DISP_EN2*, which corresponds to the ERROR subcircuit enable signal, must be pulled HIGH too.

When none of these input signals, i.e. *DISP_EN_0* and *DISP_EN_1* are HIGH, that means that the display does not have to show neither the ***Open*** nor the ***Err*** message, but the password entered by the user. Thus, *DISP_EN0*, which is the enable for the decoder of the password, must be pulled HIGH.

2.10.2 Autostart

The autostart subsystem, as the name suggests, is in charge of starting the system automatically when the simulation begins. This is needed because the ring counter that we discussed in **Subsection 2.1** needs to be pre-loaded in order to start.

In a real case scenario, a single pulse generator, such a monostable could be used to generate the required pulse. For the simulation though, it is a tad more complex, as the simulation takes some time to load, and non-linear components such as diodes tend to act in a weird way if they are powered before starting it. That is why we cannot simply connect one of the pulse generators that we discussed in **Section 2.8.2** to the *START* signal and expect it to run flawlessly.

To overcome this problem, we have simply delayed the operation of the pulse generator by performing some unnecessary actions. Since we had some pins left in the Enable Selector GAL, we decided to use it to generate said delay. In the code we can see that the GAL reads the value of the *READY* signal, which is HIGH when the simulation starts and it checks that it is indeed HIGH, and as a consequence, it pulls the *TURN_ON* HIGH. After that, the signal passes through a buffer, so as to add its propagation time to the one of the GAL, and, finally, the pulse generator is triggered, pre-loading the initial ring counter and starting the system.

This part of the code would not be needed in a real case scenario as its only function is to wait for the simulation to load properly before starting the system.

The VHDL code that describes this subsystem can be found below:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ENABLES IS
  PORT (
    CLK : IN std_logic; --CLOCK input
    READY : IN std_logic; --READY input
    DISP_EN_ : IN std_logic_vector(0 TO 1); --Display choice input
    DISP_EN : OUT std_logic_vector(0 TO 2); --Display choice output
    TURN_ON : OUT std_logic --Begin
  );
END ENABLES;

ARCHITECTURE ARCH OF ENABLES IS
BEGIN
  PROCESS (CLK)
  BEGIN
    IF (rising_edge(CLK)) THEN
      IF (READY = '1') THEN
        TURN_ON <= '1';
      ELSE
        TURN_ON <= '0';
      END IF;
      IF DISP_EN_(0) = '1' THEN --Display OPEN
        DISP_EN(0) <= '0';
        DISP_EN(1) <= '1';--Enable tristates OPEN
        DISP_EN(2) <= '0';
      ELSIF DISP_EN_(1) = '1' THEN --Display ERR
        DISP_EN(0) <= '0';
        DISP_EN(1) <= '0';
        DISP_EN(2) <= '1';--Enable tristates ERROR
      ELSE --Display RAM
```

```
33          DISP_EN(0) <= '1';--Enable decoder and tristates display
34          DISP_EN(1) <= '0';
35          DISP_EN(2) <= '0';
36        END IF;
37      END IF;
38    END PROCESS;
39  END ARCH;
```

The Proteus Subassembly of both circuits is attached below:

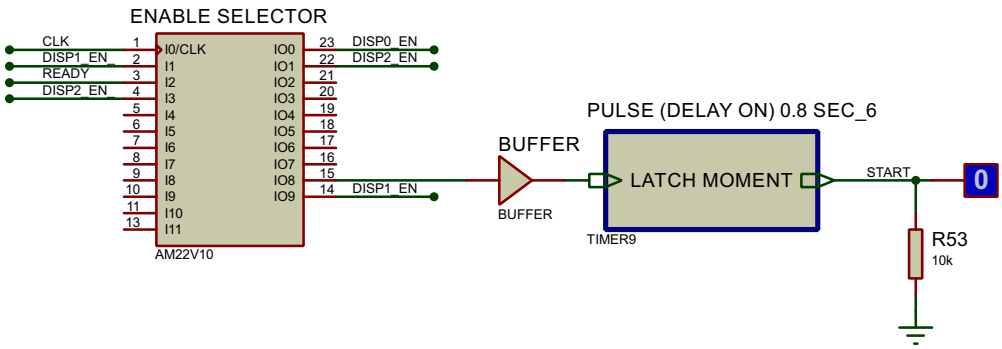


Figure 26: Enable Selector & Autostart Proteus Subassembly

Conclusion

In a nutshell, this project has posed an interesting challenge and it has given us the oportunity to not only learn about VHDL in the process but also to work with the different modules and ICs that we have seen in the course.

Out system is able to perform the duties that we initially intended it to perform even after taking into account the limitations of the hardware that we have used. The restrictions of the GAL PLDs in terms of input/output ports, memory capacity and the programming language, which was new to us in a sense, have made us use a considerable number of GALs, which only proves how far technology has come in the past few years. This project could have been implemented more easily using CPLDs or even FPGAs or, in the other side of the spectrum, a simple microcontroller.

Even though we have managed to create a fully functional safe deposit box, there are lots of things to improve nonetheless:

- Passwords can only have 4 digits, and the same number cannot be repeated consecutively.
- To introduce and validate a password one must press * and # , which makes the process slower.
- The code is most probably not as efficient as it could be. Improving it would eliminate some PLDs making the device not only more compact but also cheaper and easier to implement.
- The set password mechanism requires the user to know the binary code of the password, which is something a bit inconvenient.
- As we have said before, the use of CPLDs, FPGAs, or microcontrollers would greatly help reduce the size of the project.

All in all, we think that this project has served not only to wrap up the knowledge that we have acquired in this subject but also to learn to work as a team, mostly due to the complexity of certain parts. We sincerely hope you enjoy it as much as we have enjoyed putting it together.

References

[1] E. G. Breijo, *Lecture 5: Registers*, ser. Digital Electronics. Valencia, Spain: Universidad Politécnica de Valencia, 2020.

[2] T. L. Floyd, *Digital Fundamentals, Global Edition*, 11th ed. Harlow, United Kingdom: Pearson Education Canada, 2015.

[3] *CMOS Static RAM 16K (2K x 8 Bit)*, 6116, Rev. 1, Integrated Device Technology, Inc., Mar. 1996. [Online]. Available: https://www.princeton.edu/~mae412/HANDOUTS/Datasheets/6116.pdf.