



**UNIVERSITATEA TEHNICĂ**  
DIN CLUJ-NAPOCA

**FACULTATEA:** Automatică și Calculatoare  
**SPECIALIZAREA:** Calculatoare și Tehnologia Informației  
**DISCIPLINA:** Digital System Design  
**PROIECT:** PS2 Controller

Advisor: Blaj Ileana

Students: Muresan Victor, Pop Cristina

## TABLE OF CONTENTS

1. Specification of the project
  1. Project requirements
  2. Project considerations
2. Overview
  1. Used alphabet & Notations
  2. Black box of the design
  3. Block diagram of the components
3. Components
4. Thought process
5. Usage instructions
6. Further development
7. Annexes

## 1. Specifications of the project

### 1.1 Project requirements

Design a PS2 keyboard controller. It is required to read the keys and display the corresponding characters on the 7-segment display. The last 4 symbols will be displayed and the control keys will have special roles such as the “Enter” key displays on the board the letter pressed on the keyboard and reset button to clear all the characters from the 7-segment display, Documentation: reference manuals for FPGA boards and PS2 protocol documentation (PS2Protocol.pdf).






### 1.2 Project considerations














For this project, we worked on a Basys3 FPGA board which only has 4 anodes in the 7-segment displays, but the same implementation could be done for more than 4 anodes.














User presses a key on the keyboard and then presses Enter in order for the key to appear on the 7-segment display. When another key is entered, the previous keys are shifted to the right and the new key is displayed at the leftmost position on the 7-segment display. If the display is full or BackSpace is pressed , the display is cleared and it will show ‘0’ on all the positions.





## 2. Overview

### 2.1 Used alphabet & Notations

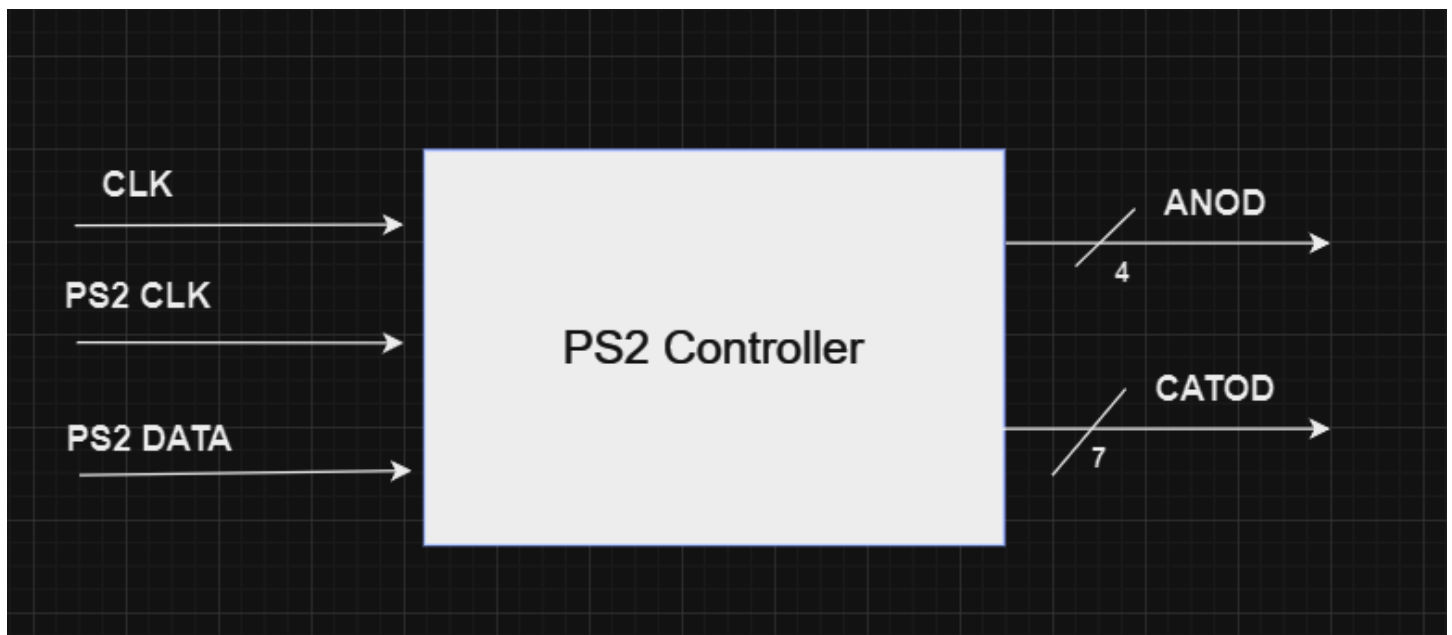
Letter	PS2 Code	Binary Code	7-segment display
A	1C	0001000	
B	32	1100000	
C	21	0110001	
D	23	1000010	
E	24	0110000	

F	2B	0111000	
G	34	0100001	
H	33	1001000	
I	43	1111001	
J	3B	1000011	
K	42	0101000	
L	4B	1110001	
M	3A	0101010	
N	31	1101010	
O	44	1100010	
P	4D	0011000	
Q	15	0001100	
R	2D	1111010	

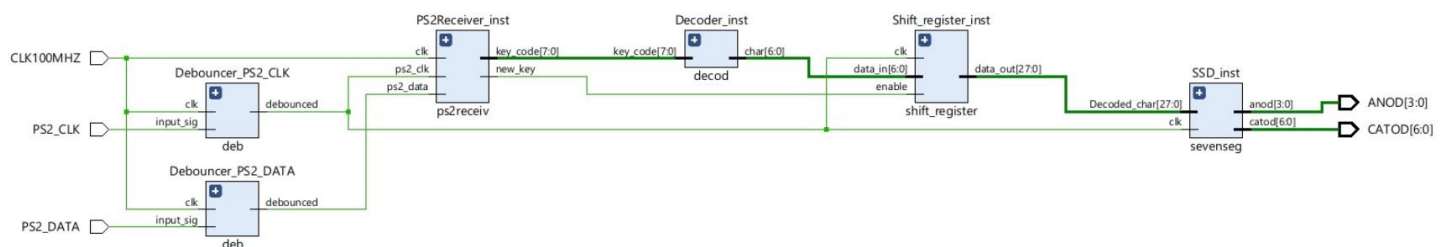
S	1B	0100101	
T	2C	1110000	
U	3C	1000001	
V	2A	1000001	
W	1D	1010100	
X	22	1101011	
Y	35	1000100	
0	45	0000001	
1	16	1001111	
2	1E	0010010	
3	26	0000110	
4	25	1001100	
5	2E	0100100	

6	36	0100000	
7	3D	0001111	
8	3E	0000000	
9	46	0000100	

## 2.2 Black box of the design



## 2.3 Schematic of our implementation



## 3. Components

### 3.1 Debouncer

Debouncing is a process used to eliminate false triggering from mechanical switches. It ensures that the input signal remains at '1' for 20 consecutive clock cycles before the output is set to '1'. If the input goes to '0' at any time then the register is reset and the counting process starts over.

```
3 |
4 | entity deb is
5 |     Port (
6 |         clk      : in  STD_LOGIC;
7 |         input_sig : in  STD_LOGIC;
8 |         debounced : out STD_LOGIC
9 |     );
10 | end deb;
11 |
12 | architecture Behavioral of deb is
13 |     signal debounce_reg : STD_LOGIC_VECTOR(19 downto 0) := (others => '0');
14 | begin
15 |     process (clk)
16 |     begin
17 |         if rising_edge(clk) then
18 |             if input_sig = '0' then
19 |                 debounce_reg <= (others => '0');
20 |             else
21 |                 debounce_reg <= debounce_reg(18 downto 0) & '1';
22 |             end if;
23 |         end if;
24 |     end process;
25 |
26 |     debounced <= '1' when debounce_reg = "11111111111111111111" else '0';
27 | end Behavioral;
```

- Debounce\_reg is a 20-bit shift register used to monitor the state of input\_sig
- When input\_sig is '1', debounce\_reg is shifted left one bit by concatenating '1' to the least significant bit
- Output is set to '1' only if all bits in debounce\_reg are '1'

### 3.2 PS2 Receiver

PS/2 protocol uses a clock signal and a data signal to transmit 11-bit frames, which includes a start bit, 8 data bits, a parity bit and a stop bit.

```

3
4 entity ps2receiv is
5     Port (
6         clk      : in  STD_LOGIC;
7         ps2_clk   : in  STD_LOGIC;
8         ps2_data  : in  STD_LOGIC;
9         key_code  : out STD_LOGIC_VECTOR (7 downto 0)
10    );
11 end ps2receiv;
12
13 architecture Behavioral of ps2receiv is
14     signal buffered : STD_LOGIC_VECTOR(10 downto 0) := (others => '0');
15     signal count : integer range 0 to 10 := 0;
16 begin
17     process (ps2_clk)
18     begin
19         if falling_edge(ps2_clk) then
20             buffered <= ps2_data & buffered(10 downto 1);
21             if count = 10 then
22                 key_code <= buffered(9 downto 2);
23                 count <= 0;
24             else
25                 count <= count + 1;
26             end if;
27         end if;
28     end process;
29 end Behavioral;

```

- Signal buffered is used to store the bits received from ps2\_data (size of 11 bits accounts for the start bit, 8 data bits, parity bit and stop bit)
- Signal count is used to keep track of the number of bits received
- On each falling edge of the clk , the current ps2\_data bit is shifted into the buffered register
- When count reaches 10, it means all of the bits have been received . The information we need is on bits 9 to 2 , excluding the start, parity and stop bit. Count is reset for the next iteration. Otherwise, count is incremented by 1 to continue the counting process

### 3.3 Decoder

```

4 entity decod is
5     Port (
6         key_code : in  STD_LOGIC_VECTOR (7 downto 0);
7         char     : out STD_LOGIC_VECTOR (6 downto 0)
8     );
9 end decod;
10
11 architecture Behavioral of decod is
12 begin
13     process (key_code)
14     begin
15         case key_code is
16             when "00011100" => char <= "0001000"; -- A
17             when "00110010" => char <= "1100000"; -- B
18             when "00100001" => char <= "0110001"; -- C
19             when "00100011" => char <= "1000010"; -- D
20             when "00100100" => char <= "0110000"; -- E
21             when "00101011" => char <= "0111000"; -- F
22             when "00110100" => char <= "0100001"; -- G
23             when "00110011" => char <= "1001000"; -- H
24             when "01000011" => char <= "1111001"; -- I
25             when "00111011" => char <= "1000011"; -- J
26             when "01000010" => char <= "0101000"; -- K
27             when "01001011" => char <= "1110001"; -- L
28             when "00111010" => char <= "0101010"; -- M
29             when "00110001" => char <= "1101010"; -- N
30             when "01000100" => char <= "1100010"; -- O
31             when "01001101" => char <= "0011000"; -- P
32             when "01010101" => char <= "1010010"; -- Q
33             when "01011011" => char <= "1101001"; -- R
34             when "01100001" => char <= "0110000"; -- S
35             when "01100011" => char <= "1010000"; -- T
36             when "01101011" => char <= "1110010"; -- U
37             when "01110001" => char <= "0100010"; -- V
38             when "01110011" => char <= "1000001"; -- W
39             when "01111011" => char <= "1100000"; -- X
40             when "10000001" => char <= "0000000"; -- Y
41             when "10000011" => char <= "0000000"; -- Z
42             when "10000101" => char <= "0000000"; -- [
43             when "10000111" => char <= "0000000"; -- \
44             when "10001011" => char <= "0000000"; -- ]
45             when "10010001" => char <= "0000000"; -- ^
46             when "10010011" => char <= "0000000"; -- _
47             when "10010101" => char <= "0000000"; -- `
48             when "10010111" => char <= "0000000"; -- {
49             when "10011011" => char <= "0000000"; -- |
50             when "10011101" => char <= "0000000"; -- ~
51             when "10011111" => char <= "0000000"; -- ~
52             when "10100001" => char <= "0000000"; -- ~
53             when "10100011" => char <= "0000000"; -- ~
54             when "10100101" => char <= "0000000"; -- ~
55             when "10100111" => char <= "0000000"; -- ~
56             when "10101011" => char <= "0000000"; -- ~
57             when "10110001" => char <= "0000000"; -- ~
58             when "10110011" => char <= "0000000"; -- ~
59             when "10110101" => char <= "0000000"; -- ~
60             when "10110111" => char <= "0000000"; -- ~
61             when "10111011" => char <= "0000000"; -- ~
62             when "10111101" => char <= "0000000"; -- ~
63             when "10111111" => char <= "0000000"; -- ~
64             when "11000001" => char <= "0000000"; -- ~
65             when "11000011" => char <= "0000000"; -- ~
66             when "11000101" => char <= "0000000"; -- ~
67             when "11000111" => char <= "0000000"; -- ~
68             when "11001011" => char <= "0000000"; -- ~
69             when "11001101" => char <= "0000000"; -- ~
70             when "11001111" => char <= "0000000"; -- ~
71             when "11010001" => char <= "0000000"; -- ~
72             when "11010011" => char <= "0000000"; -- ~
73             when "11010101" => char <= "0000000"; -- ~
74             when "11010111" => char <= "0000000"; -- ~
75             when "11011011" => char <= "0000000"; -- ~
76             when "11011101" => char <= "0000000"; -- ~
77             when "11011111" => char <= "0000000"; -- ~
78             when "11100001" => char <= "0000000"; -- ~
79             when "11100011" => char <= "0000000"; -- ~
80             when "11100101" => char <= "0000000"; -- ~
81             when "11100111" => char <= "0000000"; -- ~
82             when "11101011" => char <= "0000000"; -- ~
83             when "11101101" => char <= "0000000"; -- ~
84             when "11101111" => char <= "0000000"; -- ~
85             when "11110001" => char <= "0000000"; -- ~
86             when "11110011" => char <= "0000000"; -- ~
87             when "11110101" => char <= "0000000"; -- ~
88             when "11110111" => char <= "0000000"; -- ~
89             when "11111011" => char <= "0000000"; -- ~
90             when "11111101" => char <= "0000000"; -- ~
91             when "11111111" => char <= "0000000"; -- ~
92         end case;
93     end process;
94 end Behavioral;

```



- When key\_code changes, the case statement matches the input to these predefined values
- Key\_code is an input on 8 bits that comes from the PS/2 Receiver , each key has a code in hexadecimal which we wrote in binary. The output is a vector of 7 bits for the 7 portions of the SSD, which has been explained in the table above.

### 3.4 Seven Segment Display



The Basys3 board contains one four-digit common anode seven-segment LED display. Each of the four digits is composed of seven segments or cathodes arranged in a “figure 8” pattern, with an LED embedded in each segment.

They work using active low logic so when a cathode has a 1 value it is off and when it has value 0 it is on. Keeping this in mind we applied for each letter individually the sequence containing 7 values of 1's and 0's, based on the table provided before.

```

5 entity sevenseg is
6     Port (
7         clk      : in  STD_LOGIC;
8         Decoded_char  : in  STD_LOGIC_VECTOR (27 downto 0);
9         anod      : out STD_LOGIC_VECTOR (3 downto 0);
10        catod     : out STD_LOGIC_VECTOR (6 downto 0)
11        --new_character : in STD_LOGIC
12    );
13 end sevenseg;
14
15 architecture Behavioral of sevenseg is
16     signal refresh_count: STD_LOGIC_VECTOR(15 downto 0);
17 begin
18     process (clk)
19     begin
20         if rising_edge(clk) then
21             refresh_count <= (refresh_count + 1);
22         end if;
23     end process;
24     process(refresh_count, Decoded_char)
25     begin
26         case refresh_count(15 downto 14) is
27             when "00" => anod <="1110"; catod <= Decoded_char(27 downto 21);
28             when "01" => anod <="1101"; catod <= Decoded_char(20 downto 14);
29             when "10" => anod <="1011"; catod <= Decoded_char(13 downto 7);
30             when others => anod <="0111"; catod <= Decoded_char(6 downto 0);
31         end case;
32     end process;
33 end

```

### 3.5 Shift Register

```
5 entity shift_register is
6     Port ( data_in : in STD_LOGIC_VECTOR (6 downto 0);
7           data_out : out STD_LOGIC_VECTOR (27 downto 0);
8           enable : in STD_LOGIC;
9           clk : in STD_LOGIC);
10 end shift_register;
11
12 architecture Behavioral of shift_register is
13
14     signal aux: STD_LOGIC_VECTOR(27 downto 0) := (others => '1');
15     signal anode_count : STD_LOGIC_VECTOR(1 downto 0):="00";
16
17     begin
18     process(clk, aux)
19     begin
20     if rising_edge(clk) then
21     if enable = '1' then
22         aux<=aux(20 downto 0) & data_in;
23     end if;
24     end if;
```

The shift register is a component used for changing the anode onto which the letter is displayed on the SSD component, the register shifts the anodes from right to left, losing at each step the most significant bit and adding to the rest of the vector which contains 28 bits because each anode contains 7 bits of information. At each clock cycle when enter is pressed the display shifts the letters. This is done because the enable is linked to the pressing of the enter key.

### 4. Thought process

We used multiple sources to have a modular program that can be easily reused. Having multiple entities also makes debugging easier , as each entity can be separately tested and has clear inputs and outputs .

Our choices for the entities were made based on several aspects such as functionality, performance, reliability, decoding keyboard codes and displaying the correspondent decoded characters efficiently.

### 5. Usage instructions

Connect the board to your device and turn the switch on

Connect the keyboard through the USB port

Start pressing keys on the keyboard and you should see the key shown on one anode of the Seven Segment Display. When pressed again, the next key should be displayed on the next anode and so on. If the display is full, it will change each letter depending on the anode we're at.

## 6. Further development

One improvement we could make is to select which anode we want to update by using user input, such as 2 switches ( we have 4 anodes and we can form 4 numbers from the 2 switches interpreted in binary).

We could also extend keyboard compatibility to include more modern keyboards by implementing a USB protocol interface. This way, the user could use a wider range of devices to control the project and it would enhance user experience.

The controller could be improved by having wireless connections to eliminate the need for physical cable connections. It would offer greater flexibility and mobility in keyboard placement.