DOCUMENTATION

PS2 KEYBOARD CONTROLLER

CONTENTS

1. ASSIGNEMENT OBJECTIVES

Objective 1: Develop a comprehensive hardware interface to receive, process, and display scan codes from a PS2 keyboard using the Nexys A7 board.

PS2 Interface Design:

- Implement a PS2 receiver module in VHDL to read scan codes from the keyboard.
- Integrate debouncing logic to filter out noise from the keyboard signals.

Real-Time Data Processing:

- Implement logic to store and shift scan codes as new data is received.
- Ensure that up to 8 characters are displayed on the seven-segment display, shifting left when a new key is pressed.
- Discard repeated scan codes to prevent redundant displays when a key is held down.

System Integration:

- Write VHDL code for each component and integrate them into a single project.
- Use the Vivado design suite to simulate, synthesize, and program the design onto the Nexys A7 board.
- Test and validate the system's performance in real-time with a connected PS2 keyboard.

Objective 2: Special Key Functionality and User Interaction
Implement special roles for certain keys to enhance user interaction with the display on the board.

Backspace Functionality:

- Implement logic to delete the last character displayed when the backspace key is pressed.

Enter Key Functionality:

- Implement logic to reset the display and start a new line of input when the enter key is pressed.

Arrow Keys and Dot Positioning:

- Allow arrow keys to move a dot position indicator on the display.
- Adjust the backspace functionality based on the dot's position, ensuring that if the dot is not on the last position, the backspace deletes the character before the dot.

2. PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES

2.1 Problem Analysis

Problem Analysis involves breaking down the problem into smaller, manageable parts and understanding each component's role in the overall system. This helps in designing and implementing the solution effectively

Input Handling: Correctly receive and debounce the clock and data signals from the PS/2 keyboard.

- The PS2 receiver module captures keyboard inputs by synchronizing with the PS/2 clock (kclk) and reading data bits from the kdata line.
- It processes 11-bit frames, performs parity checks for error detection, allowing the receiving device to validate and accurately interpret each keypress.
- Use debouncing logic to filter out noise and ensure clean signal input.

Data Processing: Decode the serial data received from the keyboard into meaningful key codes.

- Implement a process to handle the falling edge of the debounced clock signal and stock data bits.
- Convert the serial data into an 8-bit key code.

Data Storage: Store the decoded key codes in a stack and manage push/pop operations.

- The stack is used because it relates to all the keyboard operations: delete is a pop, typing is a push, the pointer acts as the cursor, and typing at the cursor involves pushing the rest of the stack from the pointer.
- Implement push operations for new key codes and pop operations for backspace functionality.

Display Update: Display the key codes on a seven-segment display.

- Implement logic to update the display with the latest key code.
- Shift the displayed characters to the left when a new key code is received.
- Ensure that up to 8 characters are displayed.

Key Features and Functionalities: Implement backspace, enter, and arrow

- Backspace: Delete the last character displayed.
- Enter: Reset the display and start a new line of input.
- Arrow Keys: Move a dot position indicator and adjust the backspace functionality based on the dot's position.

By focusing on these modules, the project has an user-friendly interaction with the PS2 keyboard and seven-segment display, providing a demonstration of digital design principles using the Nexys A7 board.

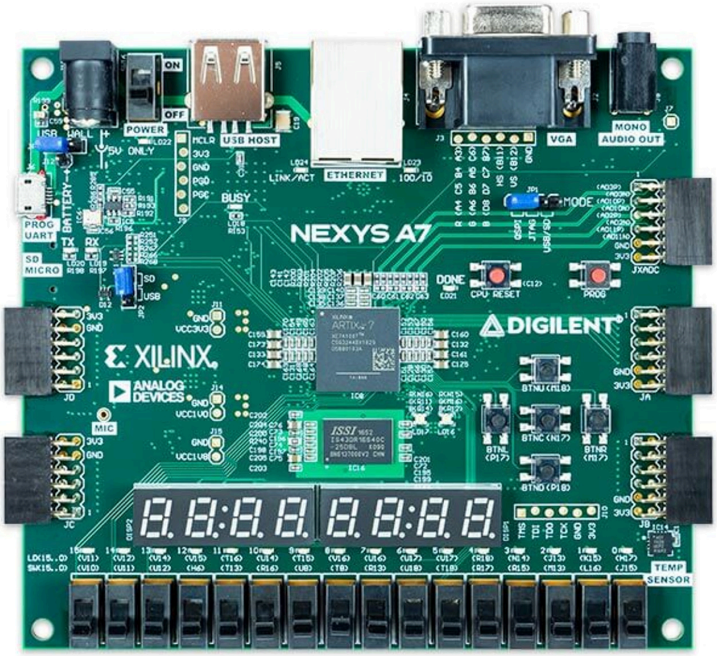
2.2 Modeling

HARDWARE PARTS

Nexys A7 Board:

Utilize the Vivado design suite for development and the Nexys A7 board for demonstration.
The Nexys A7 board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx®. With its large, high-capacity FPGA,

generous external memories, and collection of USB, Ethernet, and other ports, the Nexys A7 can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys A7 to be used for a wide range of designs without needing any other components.



PS2 KEYBOARD

The PS2 keyboard is a standard keyboard interface used for connecting a keyboard to a computer. It uses a 6-pin mini-DIN connector. The PS2 protocol is a bidirectional synchronous serial communication protocol.

Ports

PS2 Clock (kclk):

- Function: Synchronizes data transmission between the keyboard and the host. This clock signal ensures that data is transferred accurately by coordinating when each bit of data is sent.
- Type: Serial clock signal, generated by the keyboard, which signals the host to read each data bit. The keyboard sends out the clock pulses during data transmission.

PS2 Data (kdata):

- Function: Transmits data (scan codes) from the keyboard to the host. Each key press or release on the keyboard generates a specific scan code, which is sent to the host through this data line.

- Type: Serial data signal, which carries the actual information regarding which key was pressed or released. The data is sent in a series of bits, each synchronized with the clock signal.

PS2 TO USB ADAPTOR

Description:

A PS2 to USB adapter allows a PS2 keyboard to be connected to a USB port. It converts the PS2 signals to USB signals, enabling the use of a PS2 keyboard with modern devices that lack PS2 ports.

Ports:

PS2 Connector (6-pin mini-DIN):

- Function: Connects to the PS2 keyboard.
- Type: 6-pin mini-DIN.

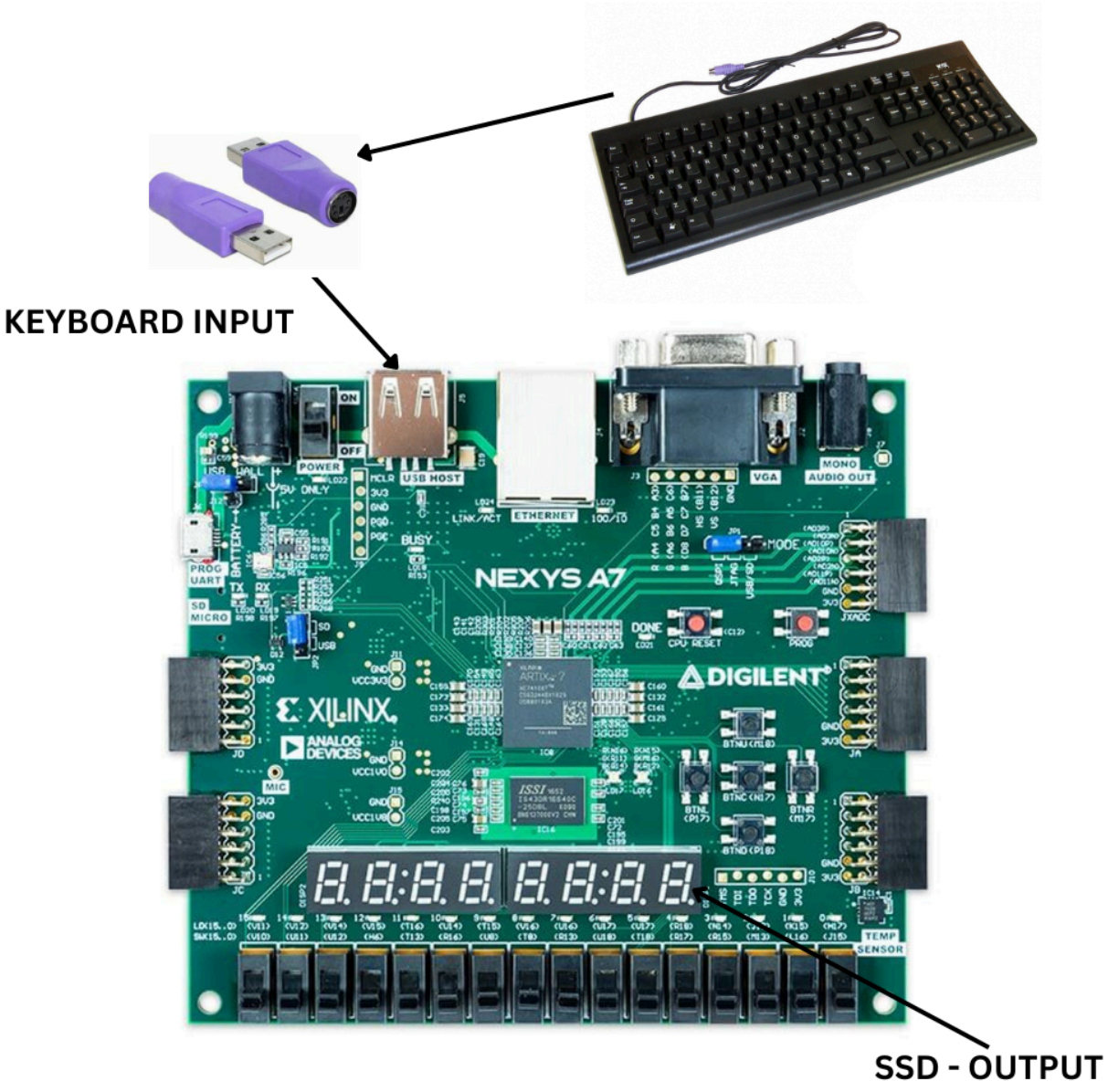
USB Connector (Type A):

- Function: Connects to a USB port on a host device.
- Type: USB Type A plug.

PERSPECTIVE FROM AFFAR

- PS2 Keyboard: A standard input device using PS2 clock and data signals for communication.
- PS2 to USB Adapter: Converts PS2 signals to USB, enabling the connection of a PS2 keyboard to a USB port.
- Nexys A7 Board: A versatile FPGA development board that interfaces with the PS2 keyboard via the PS2 to USB adapter through its USB port and dedicated PS2 clock and data pins

.

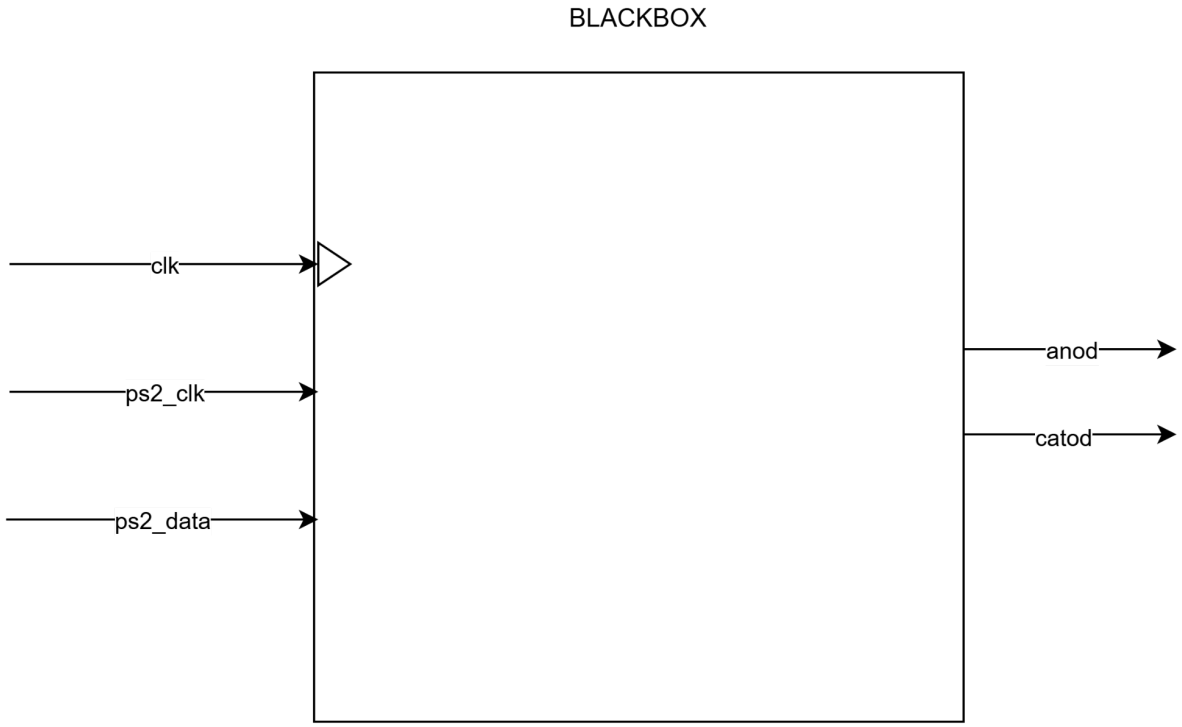


Black Box :

The PS/2 controller interface is illustrated in Figure 1, with inputs shown on the left, outputs on the right. Ports are used as follows:

PORTS THAT MUST BE CONNECTED:

- CLK - system clock input, must be 100MHz for the timing control to work properly.
- ps2_clk and ps2_data - ps2 clock and data lines, respectively. These lines should be connected to the correspondingly named pins on the board, as defined in this file.
- Anod – 7 bits.
- Catod – 7 bits



2.3 SCENARIOS

Definition:

- Scenarios are narrative descriptions of specific situations or sequences of events that illustrate how a system behaves under certain conditions.

Purpose:

- To provide detailed, contextual examples of how the system operates in real-world situations.
- To explore different possible paths and outcomes based on varying inputs or conditions.

Characteristics:

- Typically narrative and descriptive.
- Focused on a specific sequence of actions and events.
- Can include details about the environment, the actors involved, and the system's state before and after the interaction.

Scenario 1: Typing Multiple Characters

- Event: A user types multiple characters on the PS2 keyboard.
- Action: The PS2 receiver decodes each character.
- Reaction: Each character is pushed onto the stack.
- Outcome: The SSD displays the sequence of characters, with the new character added at the end.
- Next Step: The user sees the complete typed message on the SSD.

Scenario 2: Rapid Key Presses

- Event: The user rapidly presses several keys in succession.
- Action: The PS2 receiver quickly decodes each key press.
- Reaction: The debouncer stabilizes each input.
- Outcome: The stack correctly pushes each character onto the stack.
- Next Step: The SSD updates in real-time, displaying all characters without missing any input.

Scenario 3: Continuous Key Hold

- Event: The user holds down a key.
- Action: The PS2 receiver detects the initial key press and subsequent repeated signals.
- Reaction: Only the first key press is processed; repeated signals are ignored.
- Outcome: The stack pushes the character once.
- Next Step: The SSD shows a single instance of the character, regardless of how long the key is held.

Scenario 4: Using Arrow Keys

- Event: The user presses the left or right arrow key.
- Action: The PS2 receiver detects the arrow key press.
- Reaction: The stack updates the pointer position.
- Outcome: The SSD indicates the new cursor position.
- Next Step: The user can type or delete characters at the new cursor position.

Scenario 5: Pressing Enter Key

- Event: The user presses the Enter key.
- Action: The PS2 receiver decodes the Enter key press.
- Reaction: The stack resets, clearing all characters.

- Outcome: The SSD clears the display.
- Next Step: The user sees an empty display and can start typing again.

2.4 USE CASES

Definition:

- Use Cases are structured descriptions of how users (actors) interact with a system to achieve specific goals.

Purpose:

- To capture functional requirements and ensure the system meets user needs.
- To identify and document the different ways users can interact with the system.

Characteristics:

- Typically formatted in a structured template with clear steps.
- Focused on achieving a specific goal or task.
- Include actors, preconditions, main flow of events, postconditions, and exceptions.

Use Case 1: Typing Characters

Actor: User

Description: The user types characters on the PS2 keyboard, and they are displayed on the SSD.

Steps:

- The user presses a key.
- The PS2 receiver decodes the scan code.
- The stack pushes the scan code.
- The SSD displays the new character.
- The user sees the updated display.

Use Case 2: Deleting Characters

Actor: User

Description: The user presses the backspace key to delete the last character.

Steps:

- The user presses the backspace key.
- The PS2 receiver detects the backspace scan code.
- The stack pops the last character.
- The SSD removes the character from the display.
- The user sees the updated display.

Use Case 3: Moving the Cursor

Actor: User

Description: The user uses arrow keys to move the cursor to insert characters at different positions.

Steps:

- The user presses an arrow key.

- The PS2 receiver detects the arrow key scan code.
- The stack updates the pointer position.
- The user presses another key to insert a character.
- The SSD displays the character at the new cursor position.
- The user sees the updated display.

Use Case 4: Starting a New Line
Actor: User
Description: The user presses the Enter key to clear the display and start fresh.
Steps:

- The user presses the Enter key.
- The PS2 receiver detects the Enter scan code.
- The stack resets.
- The SSD clears the display.
- The user sees an empty display.

Use Case 5: Preventing Repeated Characters
Actor: User
Description: The user holds down a key, but the system registers it only once to avoid repeated entries.
Steps:

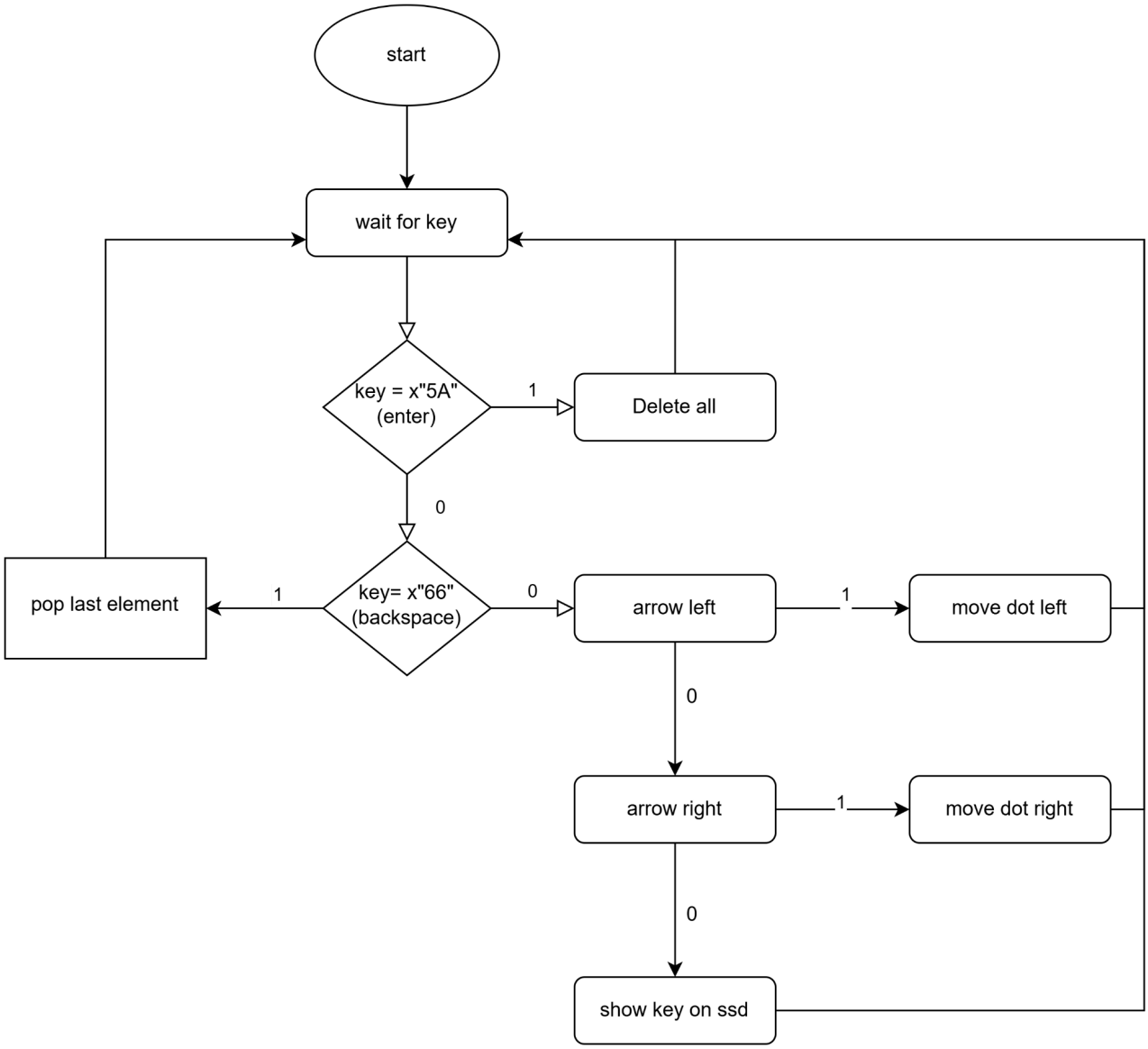
- The user holds down a key.
- The PS2 receiver decodes the initial key press.
- The stack pushes the character once.
- The SSD displays the character.
- The PS2 receiver ignores repeated signals.
- The user sees only one instance of the character.

3. DESIGN

STATE DIAGRAM

Definition:
A state diagram is a visual representation of a system that depicts the states the system can be in and how it transitions from one state to another in response to external events or conditions.

Purpose:
The purpose of the state diagram is to clearly define the behavior of the PS2 keyboard controller system. It helps in understanding how different inputs (key presses) affect the state of the system, handling keys like Enter, Backspace, and Arrow keys is well-defined. This is crucial for designing and debugging.



LOGIC DESIGN COMPONENTS

The design is made of seven main logic design components that work together to create a functional PS2 keyboard controller.

The first component is the **PS2 receiver**, which takes inputs from the PS2_Data and PS2_CLK signals and converts them into a signal that contains the last three characters written by the user. This signal is then passed on to the next components.

The next three components are **comparators** that check the last character of the signal received from the PS2 receiver. If the characters are "F0" (the key has been lifted and the design will

ignore it and stay on hold until a new key is pressed) and for special keys: enter with code "5A" and backspace with code "66".

The **multiplexor** then selects which function of the main memory component to use, depending on the key that was read.

The main **memory component** is a stack that can push, pop, reset, and hold. We chose this method for this implementation because it is similar to how a text editor writes characters. Each time a character is written, it is pushed onto the stack. Backspace is just a pop, removing the last character written. The stack has a pointer that represents the position on the seven-segment display where the point will be. This is useful when using the arrow keys to move the cursor.

The last component is a **seven-segment display** (SSD) that receives the 8-bit code from the keyboard and converts it into codes for the seven-segment display. The SSD can print all letters in the English alphabet and every decimal digit. It also decides which of the eight positions to put the dot.

4. IMPLEMENTATION

HOW TO IMPLEMENT THE LOGIC DESIGN COMPONENTS INTO THE PROJECT?

1. **File: debouncer.vhd**

Description: Stabilizes input signals from switches or buttons to prevent false detections caused by bouncing. Outputs a clean, single transition for each press or release.
COMPONENTS: Debouncer (as in the logic design component from the diagram)

2. **File: ps2receiver.vhd**

Description: This component takes inputs from the PS2_Data and PS2_CLK signals and converts them into a signal that contains the last three characters written by the user.
COMPONENTS: PS2 Receiver(as in logic design component from the diagram), debouncer.vhd

3. **File: stack.vhd**

Description: This component can push, pop, reset, and hold data. It behaves similarly to how a text editor manages characters, with operations for inserting and deleting characters.
COMPONENTS: stack (as in the logic design component from the diagram)

4. **File: ssd.vhd**

Description: This component receives the 8-bit code from the keyboard and converts it into the appropriate signals for the seven-segment display. It handles displaying characters and placing the dot.
COMPONENTS: stack(as in the logic design component from the diagram)

5. **File: ps2_controller.vhd**

Description: This file integrates all the components (PS2Receiver, stack, ssd, and comparators) to create the complete PS2 keyboard controller.
COMPONENTS: ps2.receiver.vhd, stack.vhd, ssd.vhd,
comparators, multiplexers(as in the logic design componentsfrom the diagram)

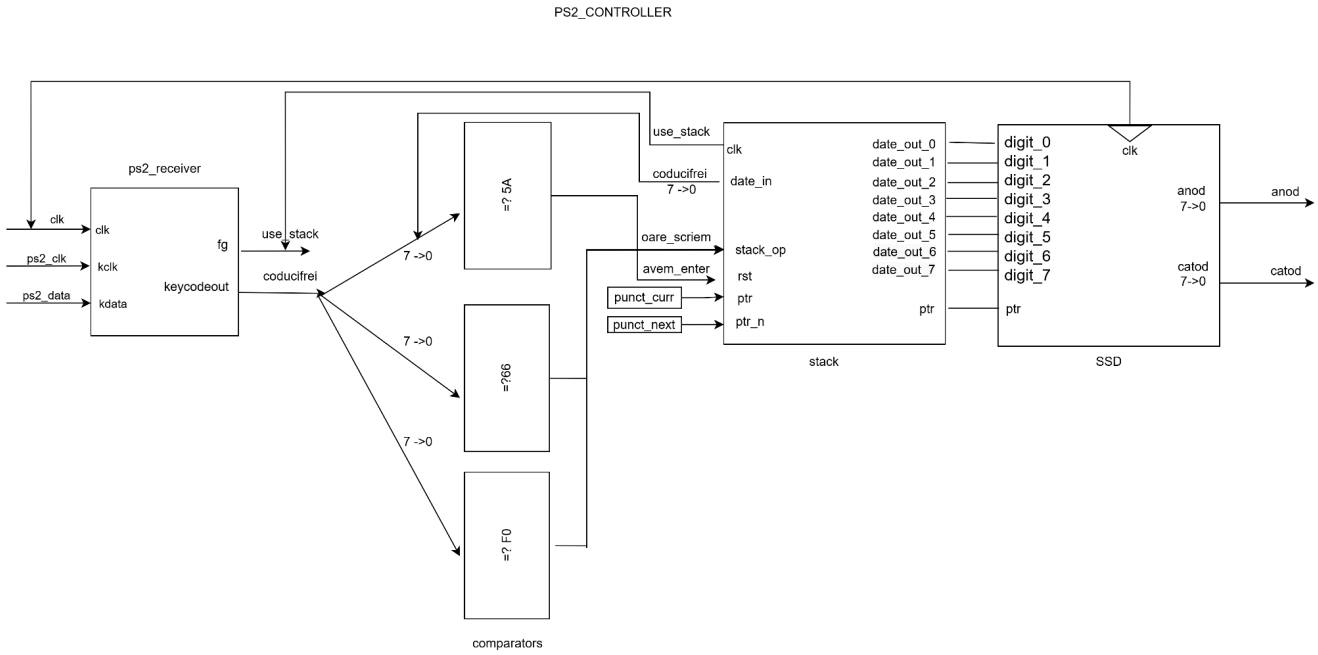
Multiplexers

Present in ps2_controller.vhd as part of the logic for selecting stack operations. This component selects which function of the main memory component (stack) to use, depending on the key that was read.

Comparators

Present in ps2_controller.vhd as part of the logic for handling special keys. These components check the last character of the signal received from the PS2Receiver to detect special keys like "F0" (key release), "5A" (enter), and "66" (backspace).

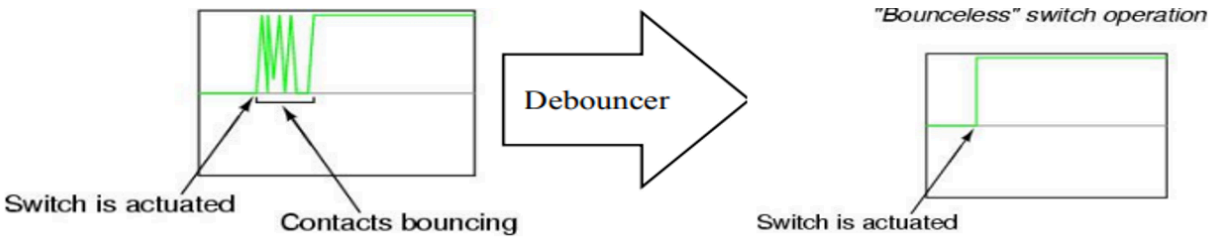
DETAIL DIAGRAM



CODE IMPLEMENTATION

1. DEBOUNCER

Knowing that due to mechanical issues the button of the FPGA board tends to bounce, giving off a unusable signal, designing a component that solves this problem was necessary.



Ports:

buton:

- Type: STD_LOGIC
- Direction: in
- Purpose: Raw input signal from the switch or button.

clk:

- Type: STD_LOGIC
- Direction: in
- Purpose: System clock signal used to synchronize operations within the debouncer module.

buton_db:

- Type: STD_LOGIC
- Direction: out
- Purpose: Debounced output signal.

```
entity debouncer is
    Port ( buton : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           buton_db : out  STD_LOGIC);
end debouncer;
```

How does it work?

Two flip-flops are connected in sequence, such that the output of the first one is connected to the input of the second one. The signal from the button is connected to the first flip-flop. The output signals from the two flip-flops are passed through a XOR gate which reset a counter on 20 bits if one of the two values of the flip-flops changes during the counted amount of time. If no changes occur, the signal is passed on via the result output port.

Architecture: Behavioral
Purpose: Contains the implementation details of the debouncer module.
Signal Declarations:

Q1:

- Type: STD_LOGIC
- Purpose: Intermediate signal used in the debouncing process.

Q2:

- Type: STD_LOGIC
- Purpose: Intermediate signal used in the debouncing process.

Q3:

- Type: STD_LOGIC
- Purpose: Intermediate signal used in the debouncing process.

```
architecture Behavioral of debouncer is
signal Q1,Q2,Q3: std_logic;
```

Process

Sensitivity List: clk

Purpose: Shifts the value of the buton signal through three intermediate stages (Q1, Q2, Q3) on each rising edge of the clock signal.

Operations:

- On the rising edge of clk:
- Q1 gets the value of buton.
- Q2 gets the value of Q1.
- Q3 gets the value of Q2.

```
process(clk)
    begin
     if (clk='1' and clk'event) then
        Q1<=buton;
        Q2<=Q1;
        Q3<=Q2;
      end if;

end process;
```

Output Assignment

Purpose: Combines the intermediate signals to produce the debounced output.

Operation: The buton_db output is set to the logical AND of Q1, Q2, and Q3. This ensures that buton_db is only asserted if the input buton has been stable for three consecutive clock cycles.

```
buton_db<=Q1 and Q2 and Q3;
end Behavioral;
```

2. *PS2_RECEIVER*

This component takes inputs from the PS2_Data and PS2_CLK signals and converts them into a signal that contains the last three characters written by the user.

```
entity PS2Receiver is
Port (  clk : in STD_LOGIC;
        kclk : in STD_LOGIC;
        kdata : in STD_LOGIC;
        fg : out STD_LOGIC;
        keycodeout : out STD_LOGIC_VECTOR (23 downto 0));
end PS2Receiver;
```

Ports
clk:

- Type: STD_LOGIC
- Direction: in
- Purpose: System clock input used to synchronize operations within the module.

kclk:

- Type: STD_LOGIC
- Direction: in
- Purpose: Clock line from the PS/2 keyboard, used to synchronize data transmission.

kdata:

- Type: STD_LOGIC
- Direction: in
- Purpose: Data line from the PS/2 keyboard, used to transmit data bits.

fg:

- Type: STD_LOGIC
- Direction: out
- Purpose: Flag to indicate the availability of new data.

keycodeout:

- Type: STD_LOGIC_VECTOR (23 downto 0)
- Direction: out
- Purpose: Output port for the keycode, holding the last three keycodes received.

Architecture Declaration

```
-- Architecture declaration of the PS2Receiver
architecture Behavioral of PS2Receiver is

    -- Declaration of the debouncer component used to stabilize the keyboard inputs
    component debouncer is
        Port ( buton : in  STD_LOGIC;
               clk : in  STD_LOGIC;
               buton_db : out  STD_LOGIC);
    end component;

    -- Signal declarations
    signal kclkf:STD_LOGIC;                -- Filtered kclk signal after debouncing
    signal kdataf:STD_LOGIC;               -- Filtered kdata signal after debouncing
    signal datacur:STD_LOGIC_VECTOR(7 downto 0); -- Current data byte being received
    signal dataprev:STD_LOGIC_VECTOR(7 downto 0); -- Previous data byte received
    signal cnt:STD_LOGIC_VECTOR(3 downto 0) := "0000"; -- Counter for bit reception
    signal keycode:STD_LOGIC_VECTOR(23 downto 0) := X"000000"; -- Storage for last three keycodes
    signal flag:STD_LOGIC := '0';           -- Internal flag to indicate data readiness
```

Debouncer Component:
Stabilizes the keyboard inputs for kclk and kdata to ensure reliable data reception.

Signal Declarations:

- kclkf: Filtered kclk signal after debouncing.
- kdataf: Filtered kdata signal after debouncing.
- datacur: Current data byte being received.
- dataprev: Previous data byte received.
- cnt: Counter for bit reception, initialized to "0000".
- keycode: Storage for the last three keycodes, initialized to X"000000".
- flag: Internal flag to indicate data readiness, initialized to '0'.

Processes

Debouncing Process

DEBOUNCE and DEBOUNCE1:
Map the debouncer component to the PS/2 clock and data lines to filter and stabilize the signals kclk and kdata received from the keyboard (ps2_clk and ps2_data from ps2_controller).

```
-- Mapping the debouncer component to the PS/2 clock and data lines
DEBOUNCE: debouncer port map(buton=>kclk, clk=>clk, buton_db=>kclkf);
DEBOUNCE1:debouncer port map(buton=>kdata, clk=>clk, buton_db=>kdataf);
```

Data Reception Process

The purpose of the Data Reception Process is to receive and assemble data bits from the PS/2 keyboard, indicate when a complete byte has been received by setting a flag, manage the bit reception counter, and update the data-ready status for further processing.

Process Trigger:

- Triggered on the falling edge of the debounced PS/2 clock (kclkf).

Bit Reception:

- Uses a case statement to handle each bit of the data byte.
- Reads each bit of kdataf into datacur based on the value of cnt.
- Sets the flag to '1' after the last bit is read, then resets the flag.

Counter Management:

- Increments the counter until it reaches 10, then resets it to "0000".

Output the Flag:

- Updates the fg output with the flag status.

```
-- Process to handle the falling edge of the debounced PS/2 clock
process(kclkf)
begin
    if kclkf = '0' and kclkf'event then  -- Detect falling edge
        -- Using a case statement to handle each bit of the data byte
        case to_integer(unsigned(cnt)) is
            when 1 => datacur(0) <= kdataf;
            when 2 => datacur(1) <= kdataf;
            when 3 => datacur(2) <= kdataf;
            when 4 => datacur(3) <= kdataf;
            when 5 => datacur(4) <= kdataf;
            when 6 => datacur(5) <= kdataf;
            when 7 => datacur(6) <= kdataf;
            when 8 => datacur(7) <= kdataf;
            when 9 => flag <= '1';         -- Set flag after last bit is read
            when 10 => flag <= '0';        -- Reset flag
            when others => null;           -- No operation for other cases
        end case;

        fg <= flag;  -- Output the flag status

        -- Manage the counter, resetting after the tenth bit
        if to_integer(unsigned(cnt)) < 10 then
            cnt <= cnt + 1;
        elsif to_integer(unsigned(cnt)) = 10 then
            cnt <= "0000";
        end if;
    end if;
end process;
```

Keycode Update Process

Process Trigger:

- Triggered on the flag set event.

Keycode Shifting:

- Ensures current data is new by comparing datacur with dataprev.
- Shifts previous keycodes and updates keycode with new data.
- Stores current data as previous data (dataprev).

Assign Keycode to Output:

- Assigns the keycode vector to the keycodeout port.

```
-- Process to update the keycode output when a new keycode is available
process(flag)
begin
    if flag = '1' and flag'event then  -- Check for flag set event
        if dataprev /= datacur then    -- Ensure current data is new
            -- Shift previous keycode and update with new data
            keycode(23 downto 16) <= keycode(15 downto 8);
            keycode(15 downto 8) <= dataprev;
            keycode(7 downto 0) <= datacur;
            dataprev <= datacur;       -- Store current data as previous data
        end if;
    end if;
end process;

-- Assign the keycode vector to the output port
keycodeout <= keycode;
```

Perspective From Afar

- Debouncing: Ensures clean input signals from the PS/2 keyboard.
- Data Reception: Handles the reception of data bits and sets a flag when new data is ready.
- Keycode Management: Shifts and updates the keycode output to store the last three keycodes
- Flagging: Indicates the availability of new data with the fg signal.

5. STACK

```
entity stack is
    Port (
        clk      : in  STD_LOGIC;
        date_in  : in  STD_LOGIC_VECTOR (7 downto 0);
        date_out0: out STD_LOGIC_VECTOR (7 downto 0);
        date_out1: out STD_LOGIC_VECTOR (7 downto 0);
        date_out2: out STD_LOGIC_VECTOR (7 downto 0);
        date_out3: out STD_LOGIC_VECTOR (7 downto 0);
        date_out4: out STD_LOGIC_VECTOR (7 downto 0);
        date_out5: out STD_LOGIC_VECTOR (7 downto 0);
        date_out6: out STD_LOGIC_VECTOR (7 downto 0);
        date_out7: out STD_LOGIC_VECTOR (7 downto 0);
        ptr_n    : in  STD_LOGIC_VECTOR (3 downto 0);
        ptr      : out STD_LOGIC_VECTOR (3 downto 0);
        rst      : in  STD_LOGIC;
        stack_op : in  STD_LOGIC_VECTOR (1 downto 0)
    );
end stack;
```

The stack entity in VHDL represents a stack data structure designed to interface with a seven-segment display. Here's a detailed breakdown of each port and its purpose:

Ports

clk:

- Type: STD_LOGIC
- Direction: in
- Purpose: System clock input used to synchronize operations within the stack module.

date_in:

- Type: STD_LOGIC_VECTOR (7 downto 0)
- Direction: in
- Purpose: Input data to be pushed onto the stack. Represents an 8-bit character or digit.

date_out0 to date_out7:

- Type: STD_LOGIC_VECTOR (7 downto 0)
- Direction: out
- Purpose: Outputs representing the top eight elements of the stack. Each port holds an 8-bit character or digit for display on the seven-segment display.

ptr_n:

- Type: STD_LOGIC_VECTOR (3 downto 0)
- Direction: in
- Purpose: Input pointer that indicates the current position for operations like push or pop.

ptr:

- Type: STD_LOGIC_VECTOR (3 downto 0)
- Direction: out
- Purpose: Output pointer that indicates the updated stack position after operations.

rst:

- Type: STD_LOGIC
- Direction: in
- Purpose: Reset signal to clear the stack and reset the pointer to the initial position.

stack_op:

- Type: STD_LOGIC_VECTOR (1 downto 0)
- Direction: in
- Purpose: Operation selector signal to determine the stack operation.
  - "10": Push operation
  - "01": Pop operation
  - "00": Hold (no operation)

```
architecture Behavioral of stack is
    type memorie is array (0 to 15) of std_logic_vector(7 downto 0);
    signal stocare: memorie := (others => x"FF");
```

Architecture Declaration
Memory Declaration:
Type: memorie is defined as an array of 16 elements, each 8 bits wide (std_logic_vector(7 downto 0)).
Signal: stocare is an instance of memorie, initialized to x"FF" (all segments off).

Process Block:
Process Trigger: The process is triggered on the rising edge of the clock (clk).

```
begin
    process(clk)
    begin
        if (rising_edge(clk)) then
```

Reset Handling:
If rst is high, the stack pointer (ptr) is reset to zero, and all elements in stocare are set to x"FF"

```
if (rst = '1') then
    ptr <= (others => '0');
    stocare <= (others => x"FF");
```

Push Operation :
(stack_op = "10"):

```
else
    case stack_op is
        when "10" => -- push
            if (date_in = x"41" and not(ptr_n = "0000")) then -- left arrow
                ptr <= std_logic_vector(unsigned(ptr_n) - 1);
            elsif (date_in = x"49" and not(ptr_n = "1111")) then -- right arrow
                ptr <= std_logic_vector(unsigned(ptr_n) + 1);
            elsif (ptr_n = "1111") then -- pointer at rightmost position
                if (not(stocare(15) = x"FF")) then -- if last element is not empty
                    for i in 0 to 14 loop
                        stocare(i) <= stocare(i + 1);
                    end loop;
                end if;
                stocare(15) <= date_in;
                ptr <= ptr_n;
            else
                -- shift elements right to make space for new data at ptr_n
                for i in 15 downto to_integer(unsigned(ptr_n)) + 1 loop
                    stocare(i) <= stocare(i - 1);
                end loop;
                stocare(to_integer(unsigned(ptr_n))) <= date_in;
                if (ptr_n /= "1111") then
                    ptr <= std_logic_vector(unsigned(ptr_n) + 1);
                end if;
            end if;
```

Left Arrow (date_in = x"41"):

- Decrement ptr if not at the leftmost position.

Right Arrow (date_in = x"49"):

- Increment ptr if not at the rightmost position.

Pointer at Rightmost Position (ptr_n = "1111"):

- Shift elements left if the last element is not empty, then insert new data at the end.

Pointer in Middle (ptr_n between "0000" and "1111"):

- Shift elements right to make space for new data at ptr_n, then insert the new data.

Pointer at Leftmost Position (ptr_n = "0000"):

- If the first element is empty, insert new data; otherwise, shift elements right and insert new data at ptr_n.

Pop Operation
(stack_op = "01"):

```
when "01" => -- pop
    if (ptr_n = "0000") then
        stocare(0) <= x"FF";
    elsif (ptr_n < "1111" and ptr_n > "0000") then
        stocare(to_integer(unsigned(ptr_n))) <= date_in;
        ptr <= std_logic_vector(unsigned(ptr_n) - 1);
    else
        stocare(15) <= x"FF";
    end if;
```

Pointer at Leftmost Position (ptr_n = "0000"):

- Clear the first element.

Pointer in Middle (ptr_n between "0000" and "1111"):

- Replace element at ptr_n with date_in, then decrement ptr.

Pointer at Rightmost Position (ptr_n = "1111"):

- Clear the last element

Output Assignments

Output Data:

The top eight elements of the stack (stocare(0) to stocare(7)) are assigned to date_out0 to date_out7.

```
date_out0 <= stocare(0);
date_out1 <= stocare(1);
date_out2 <= stocare(2);
date_out3 <= stocare(3);
date_out4 <= stocare(4);
date_out5 <= stocare(5);
date_out6 <= stocare(6);
date_out7 <= stocare(7);
```

PERSPECTIVE FROM AFAR

- Reset (rst): Clears the stack and resets the pointer.
- Push Operation (stack_op = "10"): Adds data to the stack and updates the pointer
- Pop Operation (stack_op = "01"): Removes data from the stack and updates the pointer
- Output Data: Provides the top eight elements of the stack for display.

6. *SSD - Seven Segment Display*

A **seven-segment display** is a form of electronic display device for displaying decimal numerals that is an alternative to the more complex dot matrix displays.

Seven-segment displays are widely used in digital clocks, electronic meters, basic calculators, and other electronic devices that display numerical information.

```
entity ssd is
    Port ( clk : in  STD_LOGIC;
           digit0 : in  STD_LOGIC_VECTOR (7 downto 0);
           digit1 : in  STD_LOGIC_VECTOR (7 downto 0);
           digit2 : in  STD_LOGIC_VECTOR (7 downto 0);
           digit3 : in  STD_LOGIC_VECTOR (7 downto 0);
           digit4 : in  STD_LOGIC_VECTOR (7 downto 0);
           digit5 : in  STD_LOGIC_VECTOR (7 downto 0);
           digit6 : in  STD_LOGIC_VECTOR (7 downto 0);
           digit7 : in  STD_LOGIC_VECTOR (7 downto 0);
           anod : out  STD_LOGIC_VECTOR (7 downto 0);
           catod : out  STD_LOGIC_VECTOR (7 downto 0);
           ptr : in std_logic_vector(3 downto 0));
end ssd;
```

The ssd entity represents a seven-segment display controller in VHDL. This component manages multiple digits of a seven-segment display, allowing each digit to be controlled individually.

Ports:

clk:

- Type: STD_LOGIC
- Direction: in
- Purpose: This is the system clock input used to synchronize the operations within the ssd module. It drives the timing for multiplexing the displays.

digit0 to digit7:

- Type: STD_LOGIC_VECTOR (7 downto 0)
- Direction: in
- Purpose: These inputs represent the data for each of the eight digits on the seven-segment display. Each vector contains the segment data for one digit, where each bit corresponds to a segment (a to g, and possibly the decimal point).

anod:

- Type: STD_LOGIC_VECTOR (7 downto 0)
- Direction: out
- Purpose: This output controls the anodes (or common cathodes, depending on the display type) of the seven-segment display. Each bit in this vector activates a corresponding digit. Typically, a bit set to 0 means the digit is active (active low).

catod:

- Type: STD_LOGIC_VECTOR (7 downto 0)
- Direction: out
- Purpose: This output controls the cathodes (or segments) of the seven-segment display. Each bit in this vector corresponds to a segment (a to g, and the decimal point). The combination of these bits lights up the appropriate segments to form the desired character or digit.

ptr:

- Type: std_logic_vector(3 downto 0)
- Direction: in
- Purpose: This input indicates the position of the pointer (or cursor). It is used to highlight the current active digit, often by lighting up the decimal point or some other indicator. It helps in visually tracking where the next character will be inserted or deleted.

THE COUNTER IN SSD

```
--numarator

process(clk,numar)
variable temp : unsigned(15 downto 0);
begin

if clk='1' and clk'event then
   temp := unsigned(numar) + 1;
   numar <= std_logic_vector(temp);
end if;
end process;
```

The counter in the ssd module serves a crucial role in multiplexing the seven-segment display.
How It Works: The counter (numar) is incremented with each clock cycle in the process labeled
- - numarator

Purpose

- Multiplexing:

Multiplexing is a technique that allows multiple displays to be driven by a single set of control lines by activating each display one at a time in rapid succession. This creates the illusion that all displays are lit simultaneously, even though only one display is actually active at any given moment.

- Timing Control:

The counter provides the timing control necessary to switch between the displays quickly enough that the human eye perceives them as being continuously lit.By incrementing the counter on each clock cycle, the code can systematically activate each digit for a brief period.

MUX ANOD PROCESS

```
--mux anod

process(numar)
begin
case (numar (15 downto 13)) is

when "000"=>anod<="11111110";
when "001"=>anod<="11111101";
when "010"=>anod<="11111011";
when "011"=>anod<="11110111";
when "100"=>anod<="11101111";
when "101"=>anod<="11011111";
when "110"=>anod<="10111111";
when others =>anod<="01111111";
end case;
end process;
```

The mux anod process is responsible for controlling which of the seven-segment displays (digits) is active at any given time. This is achieved by cycling through the digits rapidly, creating the illusion that all digits are being displayed simultaneously.

Purpose:

The process ensures that only one of the seven-segment displays is active at a time. This is a common technique used to multiplex the display, which reduces the number of pins required to control multiple digits.

Inputs and Outputs:

- Input: numar signal, which is a 16-bit vector used to generate a time base for multiplexing.

- Output: anod, an 8-bit vector where each bit corresponds to a seven-segment display. A bit set to 0 means the corresponding display is active.

Multiplexing Logic:
The numar signal is used to determine which digit to activate. The three most significant bits of numar (numar(15 downto 13)) are used as a selector.
The process uses a case statement to cycle through the possible values of these three bits, activating one digit at a time.

ACTIVE DIGIT MUX PROCESS

```
--mux afisare cifra activa

process(clk,digit0,digit1,digit2,digit3,digit4,digit5,digit6,digit7,numar, ptr)
begin
catod(7) <= '1';
case (numar (15 downto 13)) is
when "000"=>hex<=digit0; if( ptr = "0001" or ptr = "0000") then catod(7) <= '0'; end if;
when "001"=>hex<=digit1; if( ptr = "0010" or ptr = "0011" ) then catod(7) <= '0'; end if;
when "010"=>hex<=digit2; if( ptr = "0100" or ptr = "0101") then catod(7) <= '0'; end if;
when "011"=>hex<=digit3; if( ptr = "0110" or ptr = "0111") then catod(7) <= '0'; end if;
when "100"=>hex<=digit4; if( ptr = "1000" or ptr = "1001") then catod(7) <= '0'; end if;
when "101"=>hex<=digit5; if( ptr = "1010" or ptr = "1011") then catod(7) <= '0'; end if;
when "110"=>hex<=digit6; if( ptr = "1100" or ptr = "1101") then catod(7) <= '0'; end if;
when others=>hex<=digit7; if( ptr = "1110" or ptr = "1111") then catod(7) <= '0'; end if;
end case;
end process;
```

The process is responsible for determining which digit to display on the seven-segment display and whether to activate the dot (pointer) for that digit. Here's a detailed breakdown of how it works:

Purpose
Digit Selection: Based on the numar signal, the process selects which digit (from digit0 to digit7) to display on the seven-segment display.
Pointer Activation: The process also checks the ptr signal to determine if the dot (pointer) should be active for the currently selected digit.

Inputs and Outputs
Inputs:
- clk: The system clock.
- digit0 to digit7: 8-bit vectors representing the digits to be displayed.
- numar: A 16-bit vector used to determine the current active digit.
- ptr: A 4-bit vector indicating the position of the pointer (dot).

Output:

- catod: An 8-bit vector controlling the segments of the seven-segment display. catod(7) specifically controls the dot segment.

Process Logic

- Initial Setup: catod(7) <= '1';: The dot segment is initially turned off.

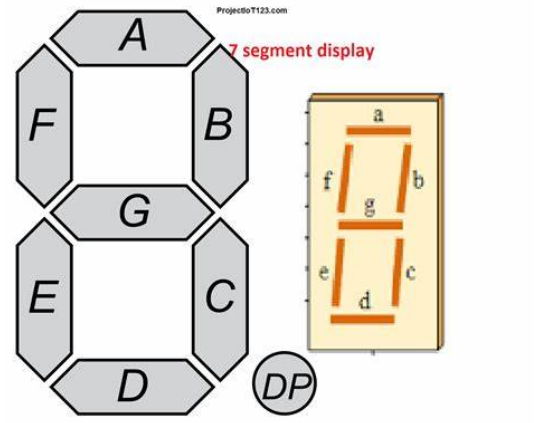
Case Statement:

- The case statement evaluates the three most significant bits of numar (numar(15 downto 13)) to determine which digit is currently active.

Digit Assignment and Pointer Check:

- For each case, the corresponding digit is assigned to hex.
- An additional check is performed to see if the ptr matches the current digit. If it does, catod(7) is set to 0 to activate the dot.

DECODER HEX TO SEVEN SEGMENT



This VHDL code segment decodes hexadecimal input (hex) to control a seven-segment display.
Each case in the hex signal sets the catod signal to light specific segments. The seven-segment display uses seven bits (a to g) to represent numbers (0-9) and letters (A-Z). Each bit in the catod vector corresponds to a segment on the display. For example, 0000001 lights all segments except g to display 0.

Code Breakdown
Process: Evaluates the hex input.
Case Statement: Maps hex values to catod bit patterns for the seven-segment display.
Bit Patterns: Each bit in catod (0 to 6) controls one of the seven segments (a to g).
Example: catod(6 downto 0) <= "0000001" lights all segments except g, displaying the digit 0. Each pattern corresponds to a specific character, enabling the display of numbers and letters.

```
process(hex)
begin
case hex is
        when "01000101" => catod(6 downto 0)<="0000001"; -- 0---NUMBERS
        when "00010110" => catod(6 downto 0)<="1001111"; -- 1
        when "00011110" => catod(6 downto 0)<="0010010"; -- 2
        when "00100110" => catod(6 downto 0)<="0000110"; -- 3
        when "00100101" => catod(6 downto 0)<="1001100"; -- 4
        when "00101110" => catod(6 downto 0)<="0100100"; -- 5
        when "00110110" => catod(6 downto 0)<="0100000"; -- 6
        when "00111101" => catod(6 downto 0)<="0001111"; -- 7
        when "00111110" => catod(6 downto 0)<="0000000"; -- 8
        when "01000110" => catod(6 downto 0)<="0000100"; -- 9
        when "00011100" => catod(6 downto 0)<="0001000"; -- A---LETTERS
        when "00110010" => catod(6 downto 0)<="0000000"; -- B
        when "00100001" => catod(6 downto 0)<="0110001"; -- C
        when "00100011" => catod(6 downto 0)<="0000011"; -- D
        when "00100100" => catod(6 downto 0)<="0110000"; -- E
        when "00101011" => catod(6 downto 0)<="0111000"; -- F
        when "00110100" => catod(6 downto 0)<="0100001"; -- G
        when "00110011" => catod(6 downto 0)<="1001000"; -- H
        when "01000011" => catod(6 downto 0)<="1111001"; -- I
        when "00111011" => catod(6 downto 0)<="1000111"; -- J
        when "01000010" => catod(6 downto 0)<="0101000"; -- K
        when "01001011" => catod(6 downto 0)<="1110001"; -- L
        when "00111010" => catod(6 downto 0)<="0010101"; -- M
        when "00110001" => catod(6 downto 0)<="0001001"; -- N
        when "01000100" => catod(6 downto 0)<="0000001"; -- O
        when "01001101" => catod(6 downto 0)<="0011000"; -- P
        when "00010101" => catod(6 downto 0)<="0010100"; -- Q
        when "00101101" => catod(6 downto 0)<="0010000"; -- R
        when "00011011" => catod(6 downto 0)<="0100100"; -- S
        when "00101100" => catod(6 downto 0)<="0110001"; -- T
        when "00111100" => catod(6 downto 0)<="1000001"; -- U
        when "00101010" => catod(6 downto 0)<="1000101"; -- V
        when "00011101" => catod(6 downto 0)<="0100011"; -- W
        when "00100010" => catod(6 downto 0)<="0110110"; -- X
        when "00110101" => catod(6 downto 0)<="1010100"; -- Y
        when "00011010" => catod(6 downto 0)<="1010100"; -- Z
        when others => catod(6 downto 0)<="1111111";
end case;
```

PERSPECTIVE FROM AFAR

- Numerator Process: Increments the numar counter for digit multiplexing.
- Multiplexer for Anode: Activates each of the 8 digits sequentially by setting the corresponding bit in anod low.
- Digit Selection Process: Selects the digit to display based on the numar counter, setting the hex value and controlling the dot based on ptr.
- Hex to 7-Segment Decoder: Converts the hex value to the corresponding 7-segment display pattern in catod.

5. RESULTS

Functional Verification

- The PS2 keyboard controller successfully captures key presses from the PS2 keyboard via the PS2 to USB adapter connected to the Nexys A7 board.
- Real-time display on the seven-segment display works as expected, showing the most recent key presses and shifting older characters to the left.

Special keys such as Enter, Backspace, and Arrow keys perform their designated functions accurately:

- Enter Key (x"5A"): Clears the display.
- Backspace Key (x"66"): Removes the last character before the dot or the last character if the dot is at the end.
- Arrow Keys (x"6B" for Left, x"74" for Right): Move the dot left or right, allowing character insertion at specific positions.

Edge Cases Handling

- The controller correctly discards repeated key codes when a key is held down, preventing duplicate entries on the display.
- Boundary conditions such as stack overflow or underflow are managed effectively:
- Overflow: When the stack is full, new entries push out the oldest entries.
- Underflow: The stack pointer does not decrement below zero, ensuring stability.

Performance Metrics

- The debouncing mechanism for the PS2 signals works efficiently, ensuring accurate key press detection without false triggers.
- The seven-segment display updates occur without noticeable delay, providing a smooth user experience.

6. CONCLUSION

The PS2 keyboard controller project demonstrates a robust design capable of interfacing a PS2 keyboard with the Nexys A7 board to display key presses on a seven-segment display. Key functionalities, including handling special keys and real-time updates, have been implemented and verified successfully. The use of a stack data structure for managing key codes provides an efficient method for text entry and

manipulation, mirroring the behavior of a simple text editor. This project showcases effective problem-solving and design skills, achieving the objectives set forth in the initial project description.

Key takeaways include:

- The importance of debouncing in signal processing for accurate data capture.
- The utility of state diagrams in planning and implementing complex control logic.
- Effective handling of edge cases to ensure system stability and reliability.

Future enhancements could include adding support for additional special keys, implementing a more sophisticated user interface, or expanding the display capabilities beyond the seven-segment display for more complex text output.

7. BIBLIOGRAPHY

https://en.wikipedia.org/wiki/Seven-segment_display
https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual
What Is a Use Case & How To Write One | Wrike
Scenario Planning in 8 Simple Steps | monday.com Blog
State Diagrams Explained: A Visual Guide to Complex Systems | Creately