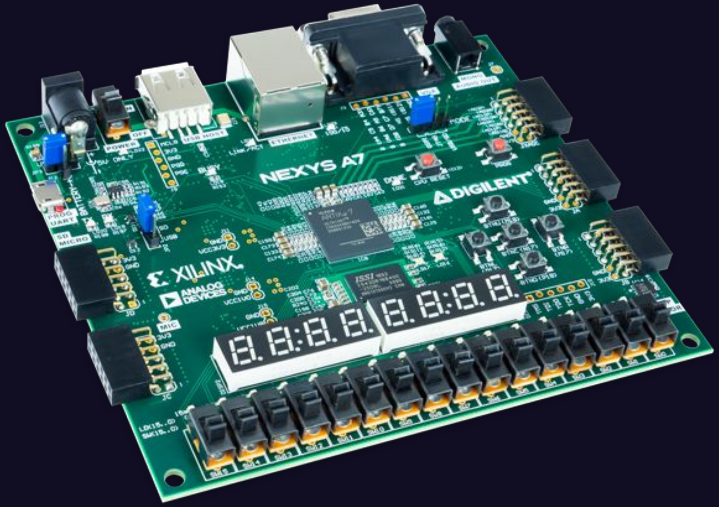
MULTI-INPUT TRANSLATOR

- By Phyliss, Kelsey, Yuxuan, and Bao (Group 18)

Overview

Designing a compact messaging system using switches for character input, a select and enable button to confirm/delete a character, accelerometer tilt inputs to control character position, and final VGA display to monitor



Goal/Motivation

What We're Doing:

- Designing a compact, hardware-efficient messaging system.
- Providing a simple way to input, edit, and store characters, displaying them dynamically on a VGA screen in real-time.

Why It Matters:

- Offers an intuitive input method using minimal hardware, replacing traditional keyboards or controllers in specific contexts.
- Demonstrates a flexible, space-saving input system for environments with limited hardware resources.

Real-Life Application Example:

- Compact remote input device for kiosks or public systems.
    - Example: Users input names or short messages (e.g., for displays, ticketing systems, or interactive kiosks).

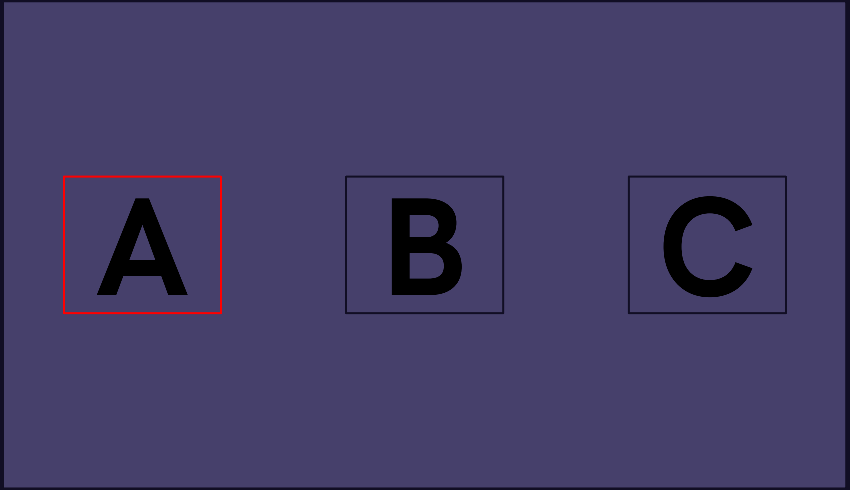
▶ Functionality

Core Features:

- Dynamically display letters on a VGA screen using bitmap representations of ASCII characters.
- Support letter insertion (en) and deletion (del) with switch-based input.
- Move a cursor-like marker to highlight the active position.

Key Focus:

- A simple, real-time system for character input, deletion, and VGA display.



Specification

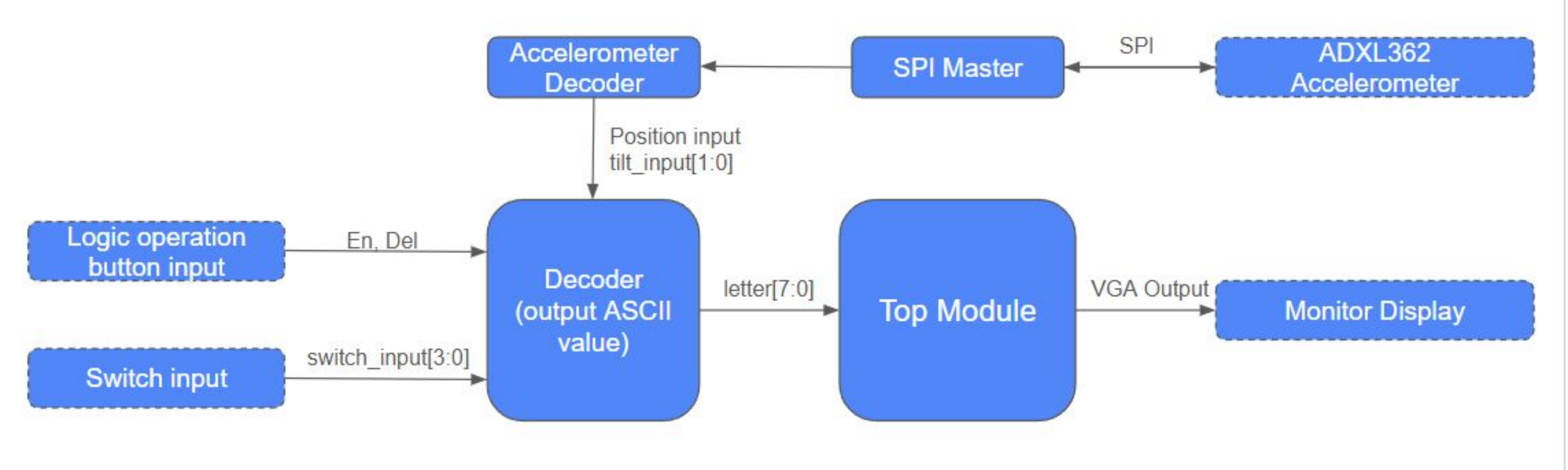
Requirements:

- Inputs:
  - Tilt input to select the active box for the next character.
  - Switches to input letters (A–Z for our implementation).
  - Delete button to remove the most recent letter.
  - Enable button to confirm a letter entry.
  - Buffer storage for up to 3 letters.
- Outputs:
  - VGA display will render three characters using bitmap representations of ASCII values.

Constraints:

- Memory limitations (3-character buffer only).
- Switch-based manual input (no tilt or dynamic input).
- VGA resolution constraints for rendering characters.
- Precise timing requirements for VGA signal generation.

Block Diagram



Successes

- Able to use bitmap representations of ASCII characters to generate VGA display
- Successful translation of ASCII binary values to ASCII characters
- Implemented basic input logic

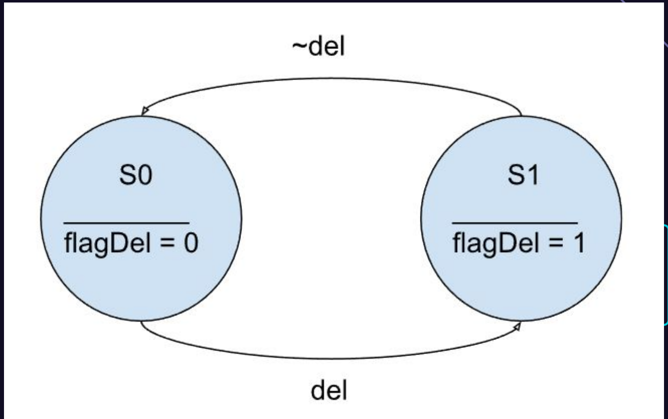
Logic Input

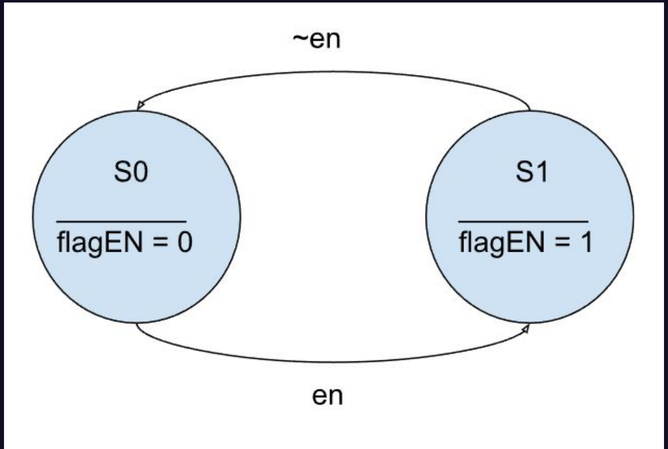
Challenge:

- Input Signals:
  - Deleting or inserting characters can occur across multiple clock cycles when the input signal remains high.

Solution:

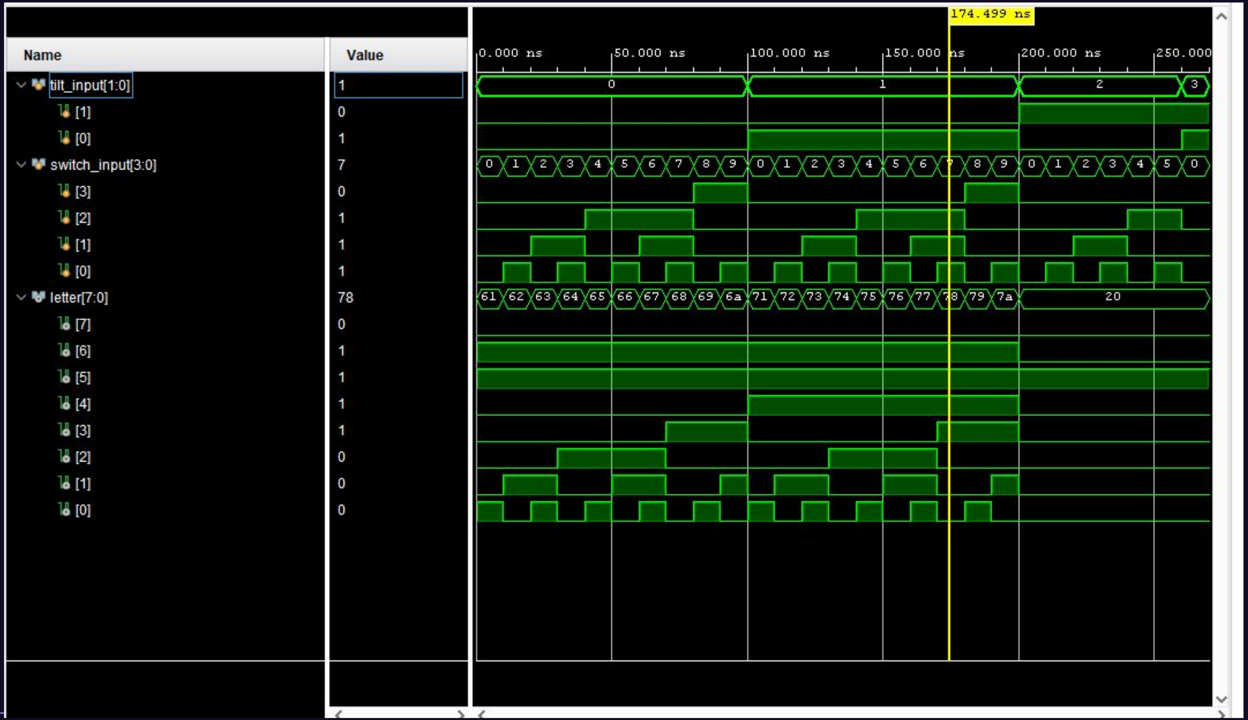
- Using Flags:
  - Introduced flags (flagDel and flagEN) to ensure single deletions and insertions per signal activation.





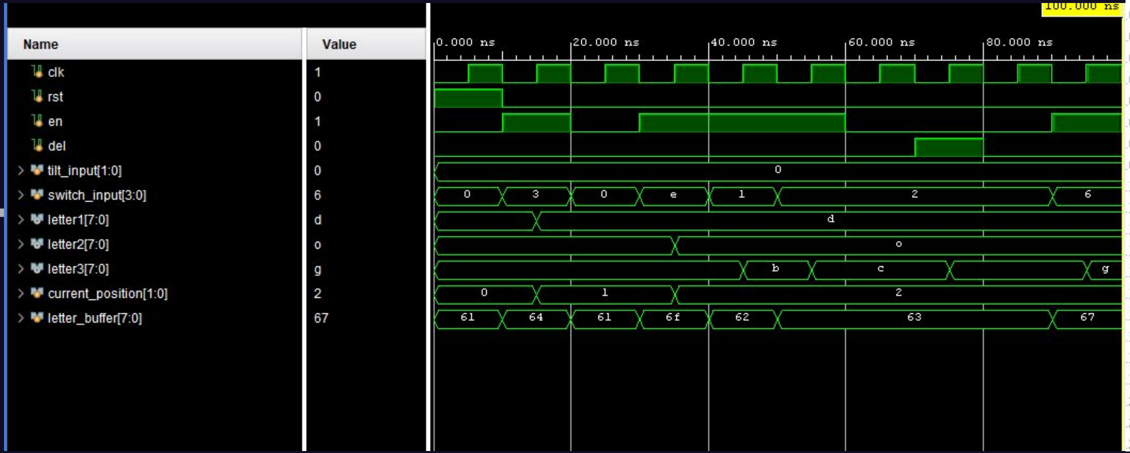
Decoder

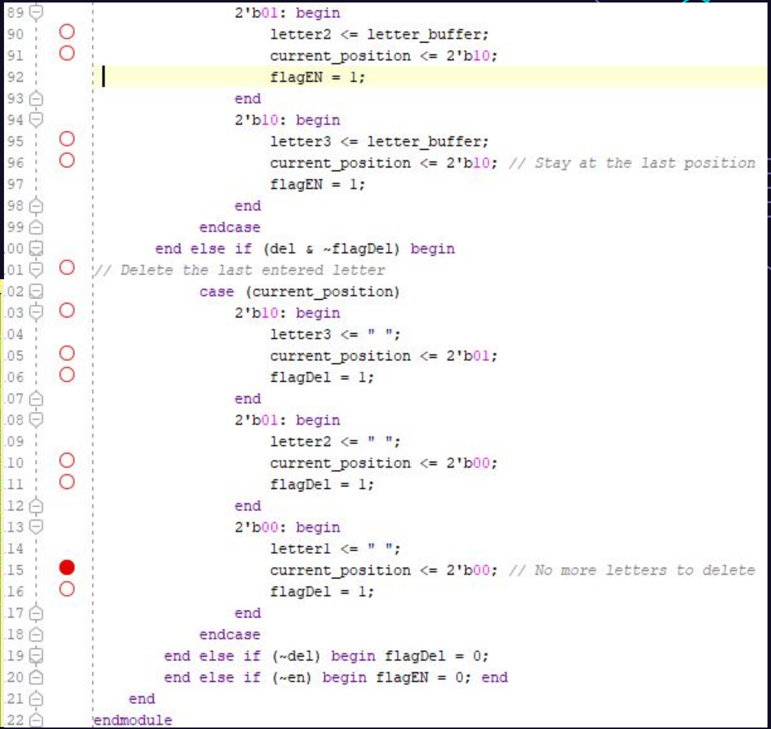
Follows ASCII table, hardcoded a to z for now.



Name	Value
tilt_input[1:0]	1
[1]	0
[0]	1
switch_input[3:0]	7
[3]	0
[2]	1
[1]	1
[0]	1
letter[7:0]	78
[7]	0
[6]	1
[5]	1
[4]	1
[3]	1
[2]	0
[1]	0
[0]	0

Decoder logic



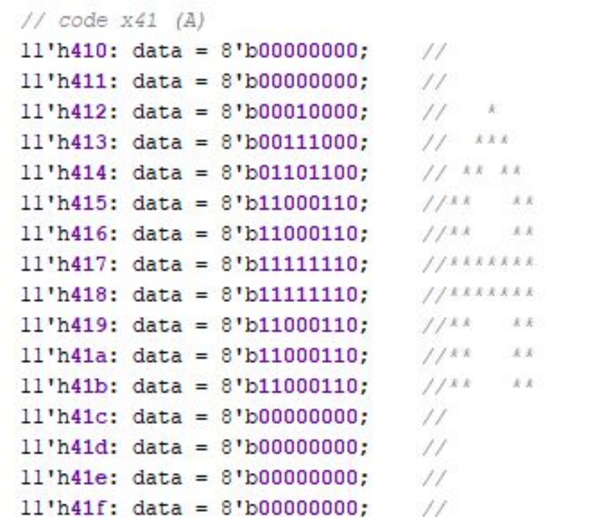


```
                2'b01: begin
                    letter2 <= letter_buffer;
                    current_position <= 2'b10;
                    flagEN = 1;
                end
                2'b10: begin
                    letter3 <= letter_buffer;
                    current_position <= 2'b10; // Stay at the last position
                    flagEN = 1;
                end
            endcase
       end else if (del & ~flagDel) begin
// Delete the last entered letter
            case (current_position)
                2'b10: begin
                    letter3 <= " ";
                    current_position <= 2'b01;
                    flagDel = 1;
                end
                2'b01: begin
                    letter2 <= " ";
                    current_position <= 2'b00;
                    flagDel = 1;
                end
                2'b00: begin
                    letter1 <= " ";
                    current_position <= 2'b00; // No more letters to delete
                    flagDel = 1;
                end
            endcase
        end else if (~del) begin flagDel = 0;
        end else if (~en) begin flagEN = 0; end
    end
endmodule
```

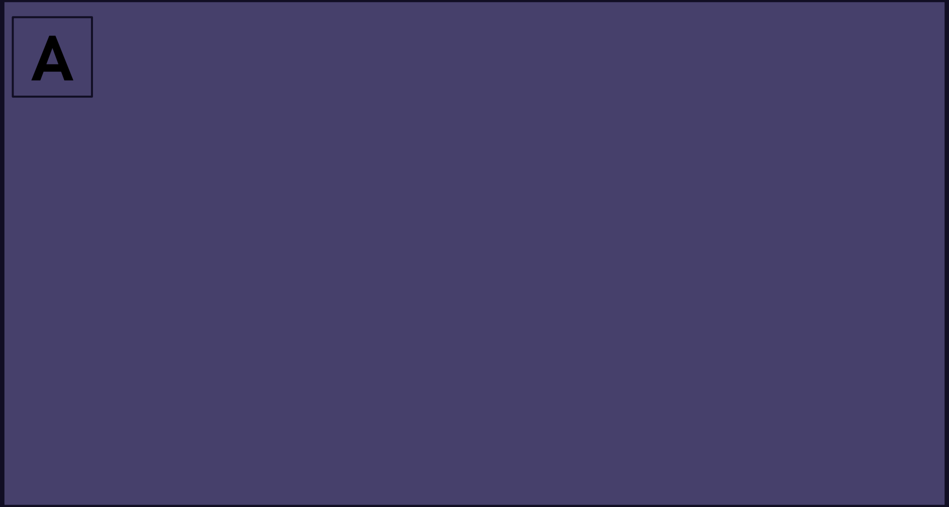
VGA Display

Letter encodings

- Challenges: size of bmap and resolution of VGA controller
- Multiple Approaches: hardcoded letters and expanded letters

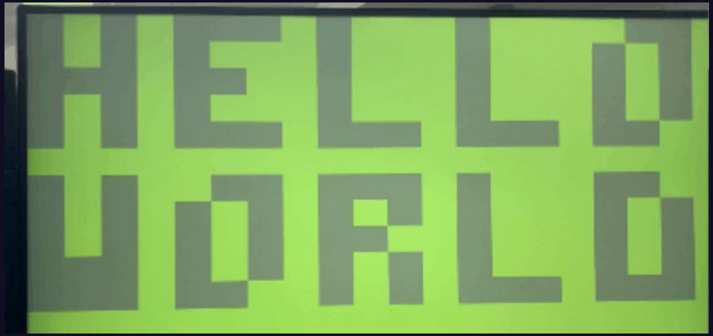


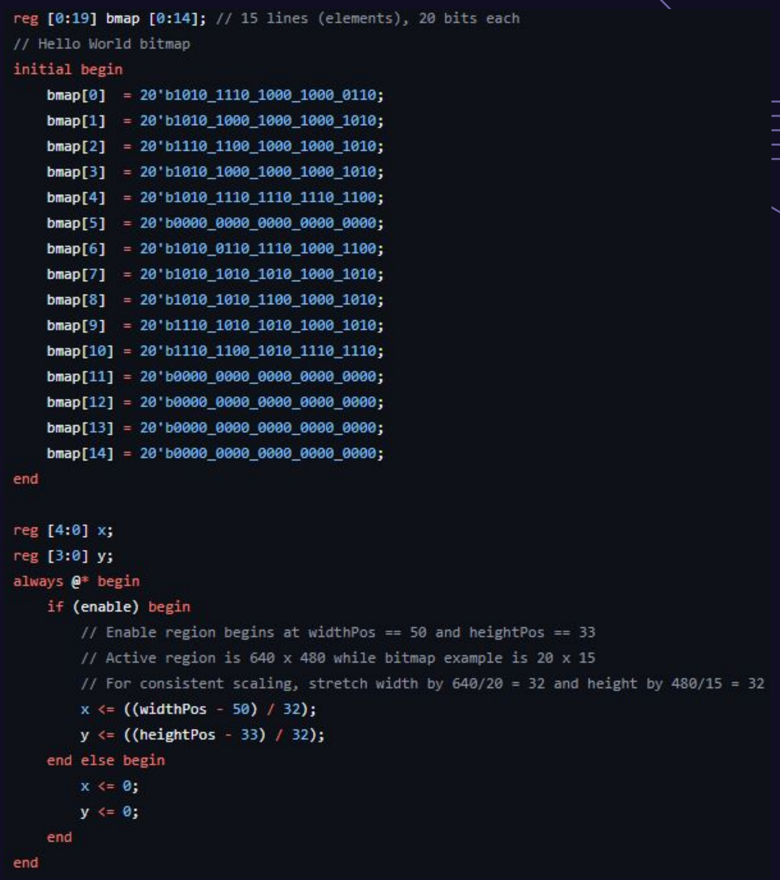
```
// code x41 (A)
11'h410: data = 8'b00000000;    //
11'h411: data = 8'b00000000;    //
11'h412: data = 8'b00010000;    //   *
11'h413: data = 8'b00111000;    //  ***
11'h414: data = 8'b01101100;    // ** **
11'h415: data = 8'b11000110;    //**   **
11'h416: data = 8'b11000110;    //**   **
11'h417: data = 8'b11111110;    //*******
11'h418: data = 8'b11111110;    //*******
11'h419: data = 8'b11000110;    //**   **
11'h41a: data = 8'b11000110;    //**   **
11'h41b: data = 8'b11000110;    //**   **
11'h41c: data = 8'b00000000;    //
11'h41d: data = 8'b00000000;    //
11'h41e: data = 8'b00000000;    //
11'h41f: data = 8'b00000000;    //
```



VGA Display (continued)

Scaling bitmap representation of “Hello World” to generate VGA output





```
reg [0:19] bmap [0:14]; // 15 lines (elements), 20 bits each
// Hello World bitmap
initial begin
    bmap[0]  = 20'b1010_1110_1000_1000_0110;
    bmap[1]  = 20'b1010_1000_1000_1000_1010;
    bmap[2]  = 20'b1110_1100_1000_1000_1010;
    bmap[3]  = 20'b1010_1000_1000_1000_1010;
    bmap[4]  = 20'b1010_1110_1110_1110_1100;
    bmap[5]  = 20'b0000_0000_0000_0000_0000;
    bmap[6]  = 20'b1010_0110_1110_1000_1100;
    bmap[7]  = 20'b1010_1010_1010_1000_1010;
    bmap[8]  = 20'b1010_1010_1100_1000_1010;
    bmap[9]  = 20'b1110_1010_1010_1000_1010;
    bmap[10] = 20'b1110_1100_1010_1110_1110;
    bmap[11] = 20'b0000_0000_0000_0000_0000;
    bmap[12] = 20'b0000_0000_0000_0000_0000;
    bmap[13] = 20'b0000_0000_0000_0000_0000;
    bmap[14] = 20'b0000_0000_0000_0000_0000;
end

reg [4:0] x;
reg [3:0] y;
always @* begin
    if (enable) begin
        // Enable region begins at widthPos == 50 and heightPos == 33
        // Active region is 640 x 480 while bitmap example is 20 x 15
        // For consistent scaling, stretch width by 640/20 = 32 and height by 480/15 = 32
        x <= ((widthPos - 50) / 32);
        y <= ((heightPos - 33) / 32);
    end else begin
        x <= 0;
        y <= 0;
    end
end
```

▶ Alternate Implementation (Tilt Module)

Key Features:

- Converts tilt values (x, y, z) to ASCII characters (A-Z).
- Uses thresholds (X_THRES, Y_THRES, Z_THRES) and incremental steps (THRES_STEP) to map tilt ranges to letters.
-
- Operates with a state machine:
  - IDLE: Waits for valid tilt input.
  - PROCESS_X/Y/Z: Determines character based on tilt axis and thresholds.
  - OUTPUT_LETTER: Outputs character (ascii_out) and asserts the valid signal.

How It Works:

- Tilt input is checked against thresholds to identify a range.
- Each range corresponds to a specific letter.
- State transitions ensure proper output of the character.

Failures

Challenges:

- XYZ tilt inputs caused unreliable character selection.
- The system consistently defaulted to first letter due to multiple matches in the state machine's for-loops.
- Threshold calibration in simulation was difficult, and state transitions caused timing issues.

Pivot:

- Switched to button-based inputs for greater stability and control.

Other Observations (During Switch Implementation):

- Deletion Logic:
  - In the testbench, the second deletion often failed because the flagDel state machine didn't reset properly, blocking further delete actions.

Citations and Resources

ASCII_ROM: Created by David J. Marion aka FPGA Dude

Project F: https://projectf.io/posts/racing-the-beam/

Nexys A7 Reference Manual:
https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual?srsltid=AfmBOooTf9fmbBJXkkVbX3Q9yLxFQurl2v2l_xBizHjNdD6rqfFJyH4J