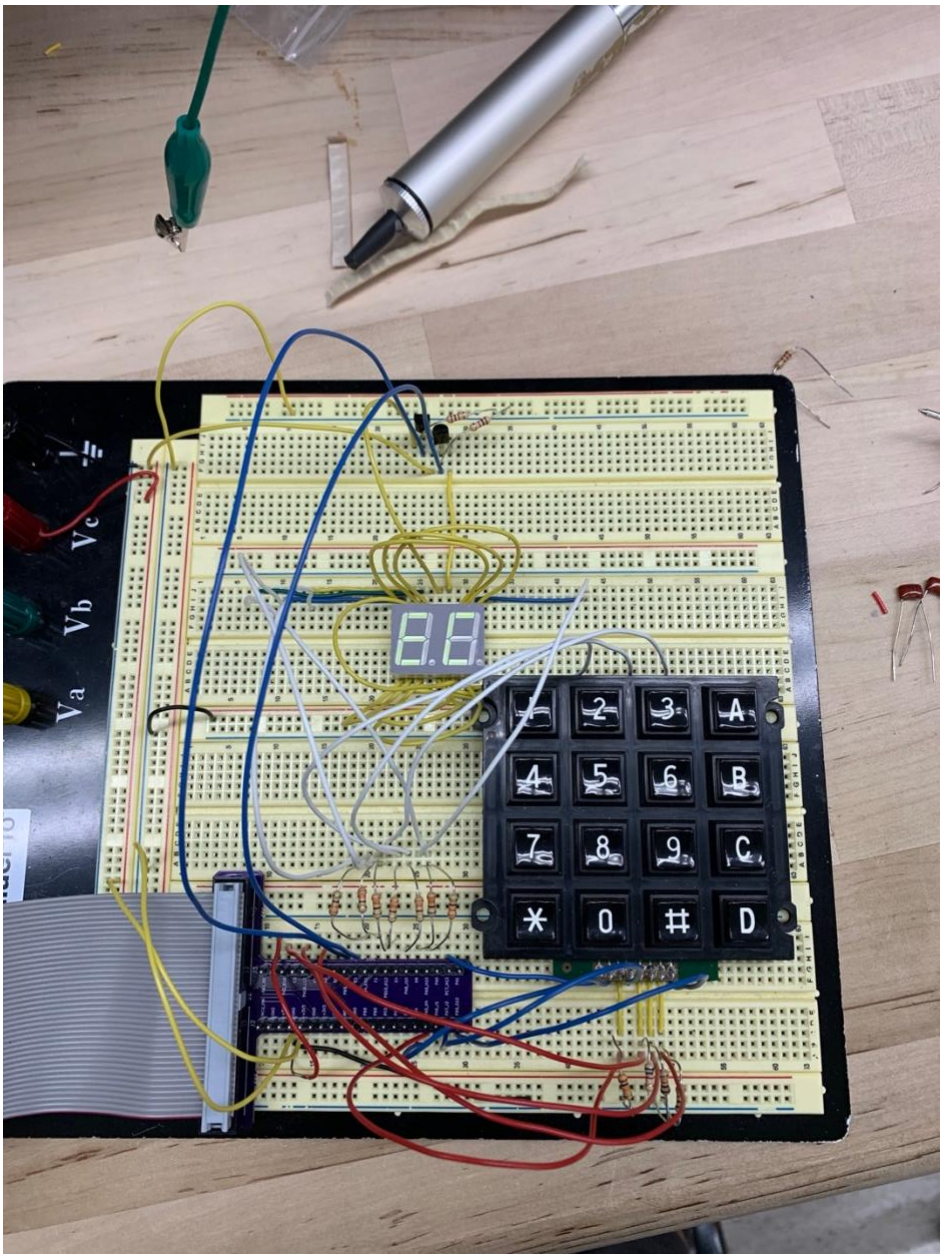
Keypad Scanner

Engineering 155 Lab IV Report
September 28th, 2021

John Hearn



Introduction:

The goal of this lab was to design a circuit interface to read a matrix keypad. We were to understand and implement a solution to deal with switch bouncing. The display was to hold two hexadecimal digits which shift to the left with each user input. The display stored these digits in registers and cycled through with user input.

Design Methodology:

Much of the hardware for this lab was reused from lab 2. The only difference was the elimination of the switches in favor of a keypad input which had eight pins: four corresponding to the rows and four corresponding to the columns, highlighted in Figure 1 below.

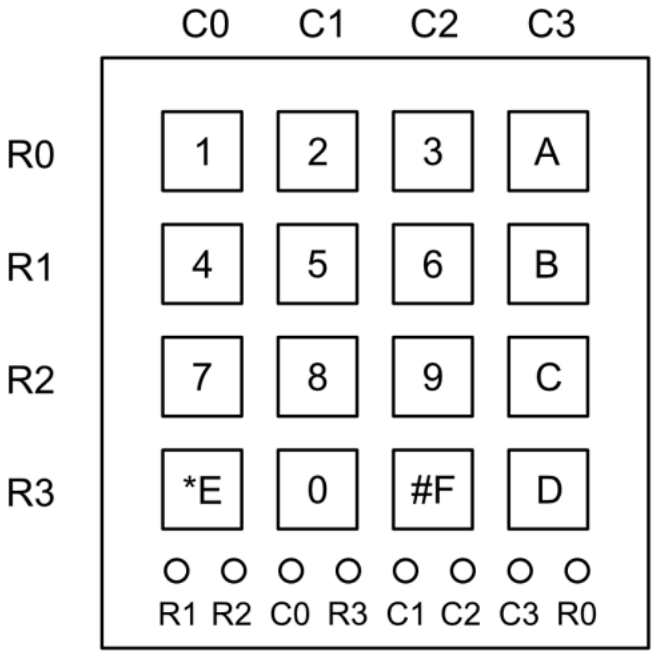


Fig. 1: Pin mapping of keypad

With no input voltage, it was necessary to use the rows as output and columns as input. The strategy was to implement a finite state machine that would alternate between powering each row waiting for a button press. Upon a button being pressed, the corresponding column pin was driven high and used as input to the board.

From this state, it was necessary to decode the input based on the coordinate of the button being pressed, outputting the corresponding number. This number was then fed into the segment assignment module written in lab 2 and shown on the dual display.

To store the digits in registers, sequential logic was used to store the numbers into their corresponding register and to shift the digits to their next register with user input.

Eliminating the issue of keybounce was a focus of this lab. I attempted to eliminate keybounce with hardware using capacitors, which inhibited by board from receiving the column input. After removing the capacitors, the system worked perfectly and there was no issue of keybounce.

If the issue of keybounce persisted, a possible solution would be to use a slower clock input to slow down the processing of input. This would make it more likely for the user's input to stop bouncing before the input was read in.

Technical Documentation:

```
module lab4_jh(input logic clk, reset,
               input [3:0] cols,
               output [3:0] rows,
               output anode1,
               output anode2,
               output [6:0] seg);

  logic powered;  // For computing which display is on
  logic [3:0] num;  // Assign the correct number to this to display
  logic [3:0] num1;
  logic [3:0] num2;



  // if powered -> turn on display 1, compute the proper digit to display
  // else turn on display 2, compute the proper digit to display
  poweredSwitch f1(clk, powered);

  // Input the rows and cols into the decoder to find which button is being pressed
  takeInput f3(powered, reset, cols, rows, num1, num2);

  assign num = powered ? num1 : num2;

  assignSegments f4(num, seg);

  assign anode1 = powered;
  assign anode2 = ~powered;
```

```
endmodule









// Using clock input determine which display should be powered
module poweredSwitch(input logic clk,
                     output logic powered);
    logic [29:0] q;
    always_ff @(posedge clk)
        q <= q + 89485;
        assign powered = q[29];

endmodule



// Compute using switch inputs which segments turn on with the display
module assignSegments(input logic [3:0] s,
                      output logic [6:0] seg);
    // Number segment display logic
    // bits go in order  gfedcba
    always_comb
        case(s[3:0])
        4'b0000: seg = 7'b1000000;
        4'b0001: seg = 7'b1111001;
        4'b0010: seg = 7'b0100100;
        4'b0011: seg = 7'b0110000;
        4'b0100: seg = 7'b0011001;
        4'b0101: seg = 7'b0010010;
        4'b0110: seg = 7'b0000010;
        4'b0111: seg = 7'b1111000;
        4'b1000: seg = 7'b0000000;
        4'b1001: seg = 7'b0011000;
```

```
      4'b1010: seg = 7'b0001000;
      4'b1011: seg = 7'b0000011;
      4'b1100: seg = 7'b1000110;
      4'b1101: seg = 7'b0100001;
      4'b1110: seg = 7'b0000110;
      4'b1111: seg = 7'b0001110;
      default: seg = 7'b1111111;
      endcase
endmodule


// FSM module for taking input
module takeInput(input logic clk, reset,
                 input logic  [3:0] cols,
                 output logic [3:0] rows,
                 output logic [3:0] currNum1,
                 output logic [3:0] currNum2);
   typedef enum logic [3:0] {s0, s1, s2, s3, s4} statetype;
   statetype state, nextState;

   always_ff @(posedge clk, posedge reset)
      if (reset) state <= s0;
      else state <= nextState;

   always_comb
      case (state)
         s0: if (cols[0] || cols[1] || cols[2] || cols[3])
             nextState = s4;
             else nextState = s1;

         s1: if (cols[0] || cols[1] || cols[2] || cols[3])
             nextState = s4;
             else nextState = s2;

         s2: if (cols[0] || cols[1] || cols[2] || cols[3])
             nextState = s4;
             else nextState = s3;
```

```
        s3: if (cols[0] || cols[1] || cols[2] || cols[3])
            nextState = s4;
            else nextState = s0;

        s4: if (cols[0] || cols[1] || cols[2] || cols[3])
            nextState = s4;
            else nextState = s0;

        default: nextState = s0;
      endcase

  assign rows[0] = (state == s4) ? 1'b1 : (state == s0);
  assign rows[1] = (state == s4) ? 1'b1 : (state == s1);
  assign rows[2] = (state == s4) ? 1'b1 : (state == s2);
  assign rows[3] = (state == s4) ? 1'b1 : (state == s3);


  // Assign the new numbers
  logic [3:0] nextNum;
  decodeInput f5(clk, reset, cols, rows, nextNum);

  always_ff @(posedge clk, posedge reset)
    if (reset) begin
        currNum1 <= 4'b0000;
        currNum2 <= 4'b0000;
    end else if ((cols[0] || cols[1] || cols[2] || cols[3]) && (state != s4)) begin  // If number input & button is no longer
pressed
        currNum2 <= currNum1;
        currNum1 <= nextNum;
    end else begin
        currNum1 <= currNum1;
        currNum2 <= currNum2;
    end

endmodule
```

```
module decodeInput(input logic clk, reset,
                   input logic [3:0] cols,
                   input logic [3:0] rows,
                   output logic [3:0] nextNum);

  always_comb
    case(cols)
      4'b0001: if (rows[0]) nextNum = 4'b0001;
               else if (rows[1]) nextNum = 4'b0100;
               else if (rows[2]) nextNum = 4'b0111;
               else if (rows[3]) nextNum = 4'b1110;
               else nextNum = 4'b0001;

      4'b0010: if (rows[0]) nextNum = 4'b0010;
               else if (rows[1]) nextNum = 4'b0101;
               else if (rows[2]) nextNum = 4'b1000;
               else if (rows[3]) nextNum = 4'b0000;
               else nextNum = 4'b0000;

      4'b0100: if (rows[0]) nextNum = 4'b0011;
               else if (rows[1]) nextNum = 4'b0110;
               else if (rows[2]) nextNum = 4'b1001;
               else if (rows[3]) nextNum = 4'b1111;
               else nextNum = 4'b0000;

      4'b1000: if (rows[0]) nextNum = 4'b1010;
               else if (rows[1]) nextNum = 4'b1011;
               else if (rows[2]) nextNum = 4'b1100;
               else if (rows[3]) nextNum = 4'b1101;
               else nextNum = 4'b0000;

      default: nextNum = 4'b0000;
    endcase

endmodule
```

Fig. 2: SystemVerilog code

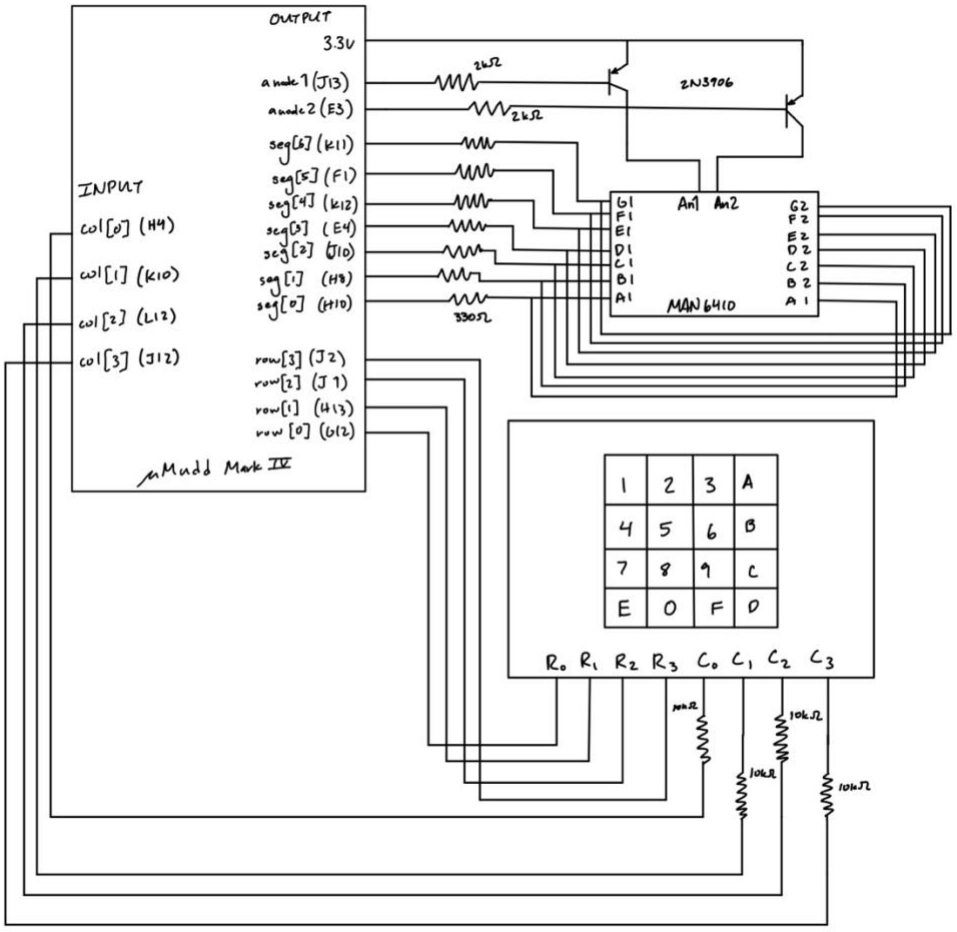


Fig. 3: Circuit Schematic

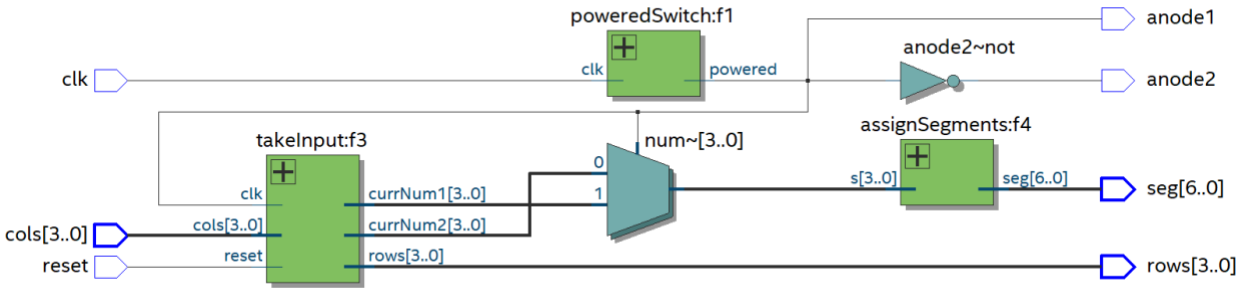


Fig. 4: RTL circuit schematic

Results and Discussion:

All of the tested inputs worked as desired and keybounce was not an issue.

If I were to redo this lab I would try using someone else's board to test first because of the issues I had with mine.

Conclusion:

I was successful in creating the keypad scanner and was able to get all the combinations of hex digits to display properly without keybounce interfering. This lab took 20 hours to complete.