Interfacing a Hex-Keypad and LCD with Krypton

EE-214: Digital Circuits Lab
4th Semester (UG Course)
Spring 2013-14

Aim:

1. To interface a 4×4 keypad matrix with Krypton, and implement switch de-bounce.
2. Display the hexadecimal value of the "key pressed" on the LEDs.
3. Display the "value" of the key pressed on a 16×2 character LCD.

Explanation:

Detecting the 'key':

A hex-keypad has 16 keys arranged in a 4×4 grid i.e., 4 rows and 4 columns. Each button is labelled with hexadecimal values from 0 to F as shown in Figure 1. There are 8 lines, corresponding to the number of rows and columns, coming from keypad which can be connected to Krypton board. On pressing a key, the row and column corresponding to that key get electrically shorted. We use this concept to determine the event "key pressed".

1	2	3	A
4	5	6	B
7	8	9	C
* (F)	0	# (E)	D

Figure 1: Hex-keypad layout

Once the event occurs, we need to detect it and take some action; in our case, we detect the 'key' and display its value on the LEDs/LCD. There are various algorithms for detecting the pressed key, we discuss only one here. However, you may explore the other ways.

We energize each column one at a time, after a pre-defined time interval and read the corresponding row value. The columns can be energized in two-ways: asserting '1' on one line and '0' on the other three, or *vice-versa*. In either of the cases, only one row line will have a defined logic level, while other three will remain floating. To overcome this, we need to either pull-up or pull-down all the lines. Pull-up requires the lines to be connected to a 3.3V source (our CPLD cannot tolerate voltage levels more than this!), but as Krypton has

only 5V supply points on-board, we choose to pull-down the lines. The circuit diagram for the same is given in Figure 2. The pull-down resistors are of 10 KΩ each.

If the row value (which was read after energizing the columns) is zero then it indicates that the 'key pressed' is not in this column. We then energize the next column, and read the row vector. This process continues until a non-zero value is read from the row. This indicates that a key could have been pressed, which we must now read and display. This non-zero row and column combination is unique for all the 16 keys and thus, we determine the 'key'.

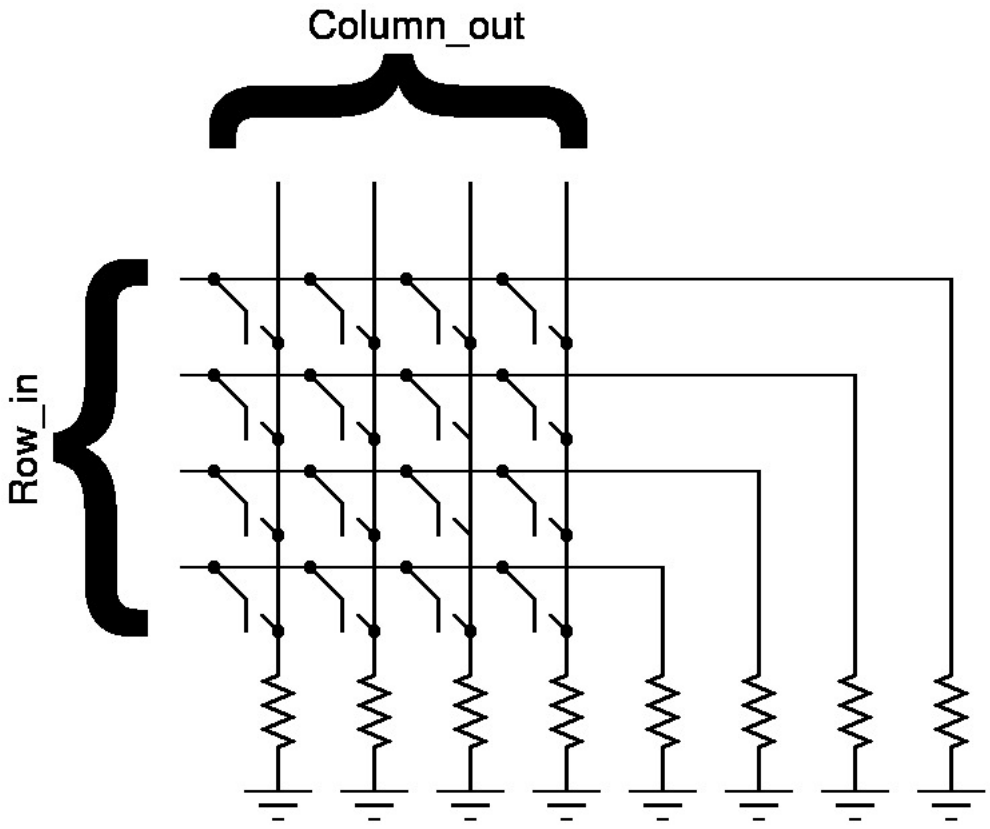

Figure 2: Circuit diagram for interfacing Hex-keypad to krypton board

The process is not as straight-forward as it seems to be. The culprit is the phenomenon of 'bouncing' of the mechanical keys. We need to overcome this i.e., we must de-bounce the switches before taking any action; else it may lead to erroneous results. The same is discussed in the following paragraph.

Switch De-bouncing:

The phenomenon of the switch 'bouncing' is illustrated in Figure 3. We ideally expect a clean transition whenever a key is pressed or released. However, such ideal cases do not exist and hence we get multiple transients on every event on the key. These transients are known to exist for about 10 ms after any event on the key. This is the maximum time for any mechanical key. Besides there may be 'false' transitions on the row lines due to various reasons like noise etc. So, we must make sure that a valid key was pressed / released before jumping to conclusions. This is called 'switch de-bouncing'.

There are many hardware and software methods to 'de-bounce' the key. We discuss one method here. The row value is read twice, after an interval of 10 ms, and the two values are compared. If found to be equal, the key is confirmed to be pressed else, we discard the row values and say that no key was pressed.

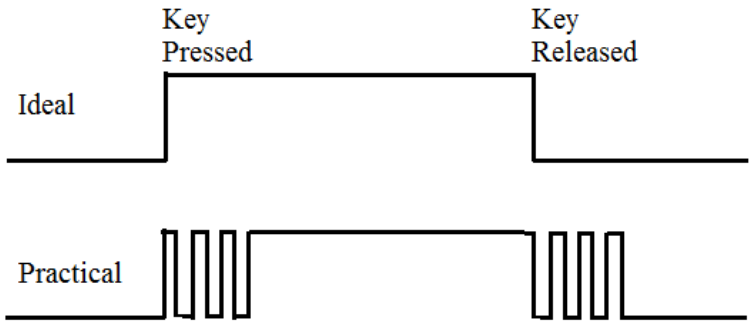

Figure 3: Switch bouncing (Illustration)

This 10 ms interval can be realized as a delay. In HDL, we cannot 'wait-for' the delay-time; this would be a non-synthesizable code. So, we count the number of clock pulses corresponding to a time of about 10 ms and thus realize the delay. The following calculation will help you to understand how to do it.

Generating 10 ms delay:

Time period of clock on Krypton board = 20 ns

No. of counts = $\frac{10\text{ ms}}{20\text{ ns}}$ = 500,000

For N bit counter:
2^N = 500,000 i.e. N ≈ 19

Delay generated = 20 ns × 2^{19} = 10.46 ms

The complete algorithm is depicted in Figure 4 and is explained as follows:

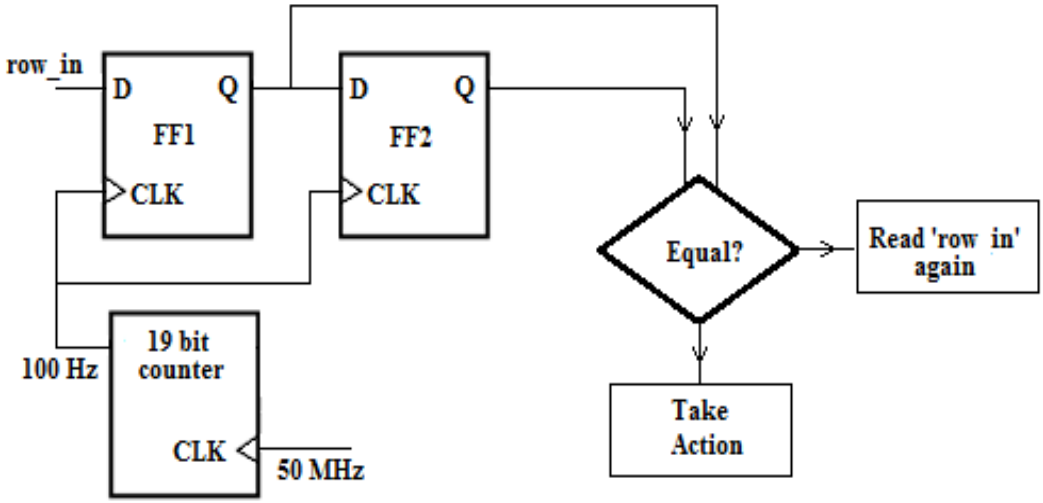

Figure 4: Algorithm for switch de-bouncing

Algorithm:

1. Load the column vector with 0001.
2. Check the row vector value, if it is zero then the pressed key is not in the column which we have energized.
3. Rotate the column vector {0001 ⟶ 0010 ⟶ 0100 ⟶ 1000}; one at a time, and again check for input row vector value; if it is non zero then a key, at the intersection of the corresponding row and column, has been pressed. If still the row vector is zero, it implies that no key was pressed and that we must return to step-1.
4. Now store the row vector value in a register and after a delay again read the row vector value.
5. Compare these two instants of row vector values; if they are same it implies the switch has been stabilized. Now the key-value can be displayed on the LEDs and LCD.

About the LCD (JHD162A):

The JHD162A is a 16 × 2 character LCD, which means that it can display 16 ASCII characters per line, over two lines. It is an 8-bit module, i.e. data may be written to or read from it through 8-bit chunks. The JHD162A has the following pins and functions-

Table I: Pins and Functions of JHD162A LCD

Pin No.	Pin Name	Function
1	GND	The power supply ground (0V)
2	VDD	The power supply voltage (5V)
3	VEE	Contrast set voltage.
4	RS	Register select input (‘0’ for command register and ‘1’ for data register)
5	RW	Read or write input (‘0’ for write and ‘1’ for read operation)
6	EN	Data latch enable. The data byte is sampled at the falling edge of this pin, and the off time needs to be at least 450ns. However, this pin should be high before the data byte (D7-D0) settles.
7 - 14	D0 – D7	Data bits, D7 being MSB and D0 being LSB.
15	VLED+	Positive supply for backlight (not used)
16	VLED-	Negative supply for backlight (not used)

The tasks to be done in the code are-

1. Display Initialization: The LCD needs to be initialized for a certain display format by appropriate commands. These commands include number of lines to be displayed (1 or 2), display start position, cursor options etc. Every time a command is sent, it must be latched on D7-D0. As we are writing a command, RW must be set to ‘0’ (i.e. write operation) and RS also to ‘0’ (i.e. command register select). The enable pin (EN) must be initially zero, set to high (for at least 500 ns), and again zero.

Usually, the following HEX commands in this order are used:
0×38- Set as two-line, 5 × 7 matrix display
0×01- Clear display
0×0E- Display on, cursor on
0×80- Force cursor to start from the first line beginning
0×06- Increment cursor (shift to right)

2. Data Write operation: We need to send the ASCII values of the characters we want to display, on D7-D0. This time, we write to the data register, i.e. RS = 1. The same timing constraints apply for EN.
3. Delay Generation: To meet the timing constraints as discussed above, a delay needs to be generated. Assume that for a large and safe margin, we select a delay of 1ms. With this delay, we take care of the constraints for the EN pin as well as the busy time of LCD. Krypton has a 50MHz clock on-board, and we will use this to generate a 1ms delay in a manner similar to the way we generated a 10 ms delay for switch de-bouncing.

To do the entire task of initialization and character-wise display step-by-step, it is best to follow a state machine approach. Consider the LCD controller as a finite state machine (FSM), comprised of a number of states. A state transition will take place ideally at the end of every 1ms delay; but since we are already waiting for 10 ms to authenticate the data, we can wait for the same time to make state transitions. Consider writing the command 0×38 to the LCD, this can be done through 3 states viz., enable pin initially zero, set to high and zero again (each for 10 ms). Now consider writing the other 4 commands, it will take in all 15 states to do the complete process. The number of states can however be reduced to 3 by defining all the commands in an array and indexing the successive commands after completion of the previous 3 states!

You must be careful while connecting the LCD, as a wrong connection may damage it! The correct connections can be done by connecting the LCD to the female header such that its pin-1 is near to the pin-1 of header-1.

Now, as you are equipped with the material required to learn switch de-bouncing and LCD initialization, you can easily proceed to fulfilling the aim of the experiment. However, some hints are given below, should you get stuck somewhere.

Hardware Description in Verilog:

```
module key_lcd (lcd, led, col, lcd_rs, lcd_rw, lcd_en, row, clk, rst);

// It is a largely followed practise to name outputs first.

output reg [7:0] lcd;               // Variable names are self-explanatory
output reg [4:0] led;
output reg [3:0] col;
output reg lcd_rs, lcd_rw, lcd_en;

// Outputs are registered

input [3:0] row;
input clk, rst;

reg [18:0] count;             // Counter for 10 ms delay
reg [7:0] temp, lcd_dat;      // temp = row-column combination of 'the key'
reg [3:0] temp_row;           // Stores row value before de-bouncing
reg [2:0] state;              // FSM to send Commands and Data to LCD
reg [2:0] count_cmd;          // Keeps count of commands sent to the LCD
reg [1:0] state_debounce;     // FSM for switch de-bouncing

// ... Declare an array to store the LCD commands
// ... Assign values to these array elements

// ... Define states for sending LCD commands and data
// We have used states S0 to S2 for sending commands,
// and S3 to S5 for sending data.

// ... Define states for switch de-bouncing
// We have named these states D0 to D2.


always @(posedge clk)
begin

      // ... Implement the count for the 10ms delay

end


always @ (???)
// ... Decide the sensitivity list
begin
      // ... Initialize State and column and LED values

            case (state_debounce)
            D0:               // Wait until non-zero row value is detected
            begin
                  if (row == 0)
                  begin
                  // ... Rotate the column and Decide the next state
                  end

                  else
                  begin
                  // ... Store the row value and Decide the next state
                  end
            end
```

```
            D1:               // Verify the row value (De-bounce)
            begin
                  // ... If row values are equal, store the column-row
                  //     combination in the variable ‘temp’
                  //     and Decide the next state

                  //... If not equal, then Decide the next state
            end

            D2:
            begin
                  //... Wait until key is released to avoid multiple
                  //    detections and Decide the next state

                  //... When key is released, Decide the next state
            end

            default:    // Why should you have a default case? Think.
            begin
                  state_debounce <= D0;
            end

            endcase
end


always @ (???)
// ... Decide the sensitivity list

begin
      case (temp)
            // ... Depending on the unique row-column combination,
            //     decide value to be sent to the LED/LCD.

            // For example: if key ‘1’ is pressed,
            // led <= 5'h01;
            // lcd_dat <= 8'h31;    ... ASCII value of ‘1’
            // and so on...

            // Can only ASCII values be sent to LCD? Think.

      default:
            // What should your LED show when no key is pressed?

      endcase

end


always @ (???)
// ... Decide the sensitivity list

begin

      if (rst)
      begin
            state <= S0;
            count_cmd <= 3'h0;      // Index to the LCD command array
      end
```

```
else
begin

      case (state)

      S0:         // S0 to S2: Send LCD commands
      begin
            if (count_cmd < 3'h5)
            begin
                  lcd <= lcd_cmd [count_cmd];
                  lcd_rs <= 1'b0;
                  lcd_rw <= 1'b0;
                  lcd_en <= 1'b0;
                  state <= S1;
            end

      // LCD requires 5 commands to be sent.
      // We send these commands and wait for valid LCD data

            else
                  state <= S3;
      end

      S1:
      begin
            lcd <= lcd_cmd [count_cmd];
            lcd_rs <= 1'b0;
            lcd_rw <= 1'b0;
            lcd_en <= 1'b1;
            state <= S2;
      end

      S2:
      begin
            lcd <= lcd_cmd [count_cmd];
            lcd_rs <= 1'b0;
            lcd_rw <= 1'b0;
            lcd_en <= 1'b0;
            state <= S0;
            count_cmd <= count_next_cmd;
      end

      S3:               // S3 to S5: Send LCD data
      begin
            lcd <= lcd_dat;
            lcd_rs <= 1'b1;
            lcd_rw <= 1'b0;
            lcd_en <= 1'b0;
            state <= S4;
      end

      S4:
      begin
            lcd <= lcd_dat;
            lcd_rs <= 1'b1;
            lcd_rw <= 1'b0;
            lcd_en <= 1'b1;
            state <= S5;
      end
```

```
            S5:
            begin
                  lcd <= lcd_dat;
                  lcd_rs <= 1'b1;
                  lcd_rw <= 1'b0;
                  lcd_en <= 1'b0;

            // Monitor the state of de-bounce FSM to avoid displaying
            // the same character multiple times due to long key-press.

                  if (state_debounce == D1)
                                          state <= S3;
                  else
                                          state <= S5;
            end

            default:
            begin
                  state <= S0;
                  count_cmd <= 3'h0;
            end

            endcase

      end

end

endmodule
```

NOTE: When you ‘reset’ the CPLD, in which state do you expect it to wake-up? Initialize the required values accordingly...