BEE 271 Digital circuits and systems
Summer 2016
Lab 3: Keypad scanner

1 Objectives

In this lab, you will design and build a keypad scanner and display as shown in figure 1.

1. The display will use the 7-segment decoder from lab 2.
2. The keypad is connected via the GPIO (general purpose I/O) connector.
3. The * (asterisk) key should mean hex E and the # (pound sign) key should mean hex F. All the other keys should be as marked.
4. When a key is pressed, the new digit should be displayed in the rightmost 7-segment display. If no key is being pressed, the display should be blank.
5. A count of the cumulative number of keystrokes should always be displayed in the four leftmost 7-segment displays with leading zero suppression.
6. The remaining 7-segment display should always be blank.
7. One of the pushbutton switches should provide a reset function, blanking all the digits.

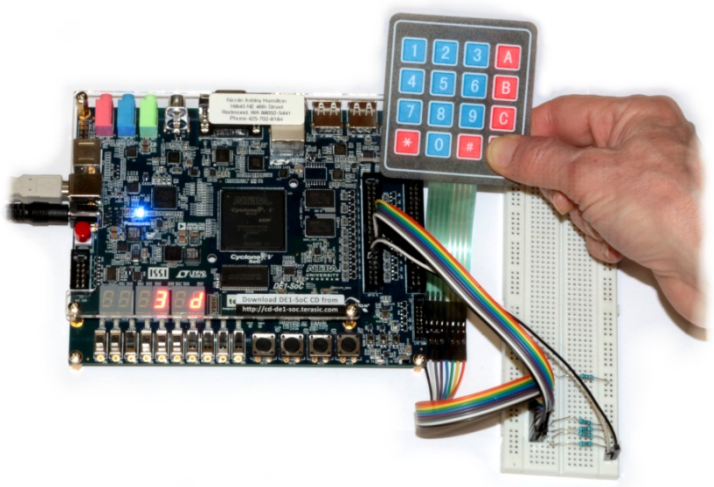

Figure 1. When a key is pressed, it's displayed and the count is incremented

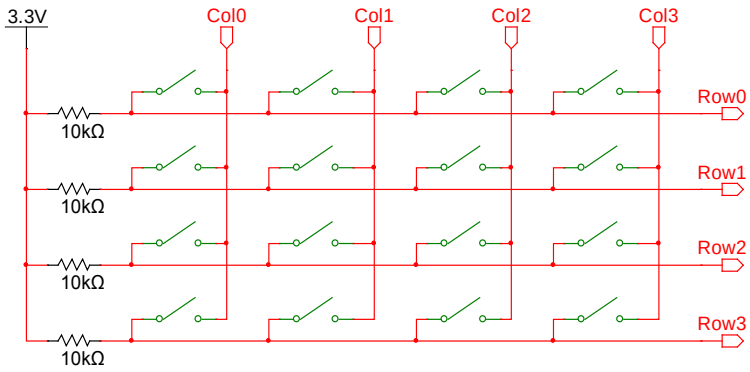

Figure 2. Keypad with pull-up resistors. The columns are the inputs and the rows are the outputs.

By the end of the lab, you should be comfortable designing, building and debugging a useful clocked sequential circuit in Verilog and you will have learned how a classic problem of working with mechanical switches is solved.

2 Work product

At the end of this lab, you must demo your design and submit your code as a .v file. That is all you need to submit. You will not be writing a report.

3 Keypad scanning

It's impractical to run even one wire per key to any large keyboard, so the standard solution for decades has been to arrange the switches into columns and rows as shown in figure 2. The coordinates for any given key are referred to as the *scan code*, which is then mapped to the appropriate *character code*.

Using this technique, the number of wires required, n_w, grows only with the *square root* of number of keys, n_k, not linearly.

$$n_w = 2\sqrt{n_k}$$

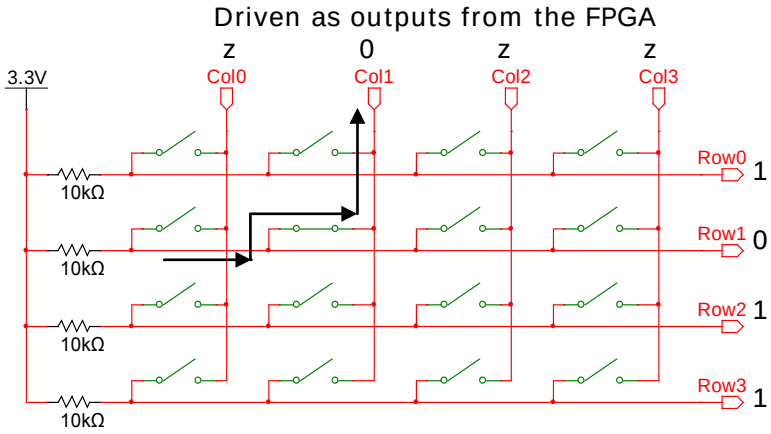

Figure 3. A closed switch creates a path to pull a row to 0 with only 330 µA.

For a keypad with 16 keys, this means we need only 8 wires, not 17 wires (one for each key + a common wire.)

Each row has a weak pull-up, meaning a resistor tied high to guarantee the rows will normally be a logic level 1, not floating, but with a high enough value resistor that it won't take much current to pull the row to 0. Using 10 KΩ resistors, we know from Ohm's Law that it will take only 330 µA to pull a row to ground.

$$I = \frac{E}{R} = \frac{3.3\ V}{10\ K\Omega} = 330\ \mu A$$

When a key is pressed, it connects a row wire to a column wire. By scanning the columns very rapidly, pulling just one column at a time to 0 while putting 1s on all the other columns as shown in figure 3, we can quickly discover any key that's pressed because when we pull its column to 0, the row it's connected to will also go to 0.

4 Procedure

In this lab, you'll scan the keypad, identify when a key is pressed and display it. You'll also display a running count in hex of the number of key depressions of same key in a row. What you'll likely discover is that sometimes when you press a key, it works and the count only goes up by only one as it should. But often the count jumps by 2 or 3 because the key is bouncing.

In the next lab, you'll add logic to debounce the keypad to ensure that if you press a key once, you get exactly one key depression.

Here are the design steps you'll follow.

Stage 1: Simple keypad scanner

1. Use the continuity test feature of the multimeter to verify that the 8 pins of the keypad are wired as expected, left-to-right as rows 0 to 3, then columns 0 to 3 as shown in figure 5.
2. Verify that you've identified the 3.3 V and ground pins on the GPIO header with the multimeter.
3. Breadboard the circuit with the keypad hooked to odd-numbered GPIO pins 11 through 25 with pull-up resistors to 3.3 V on the rows, circled in figure 6. *Row[0] on the keypad should be connected to GPIO[25].*
4. Use the System Builder tool and Quartus Prime to create an empty Verilog project with the necessary inputs and outputs.
5. Add your seven segment decoder module.

Figure 5. Keypad pinout.

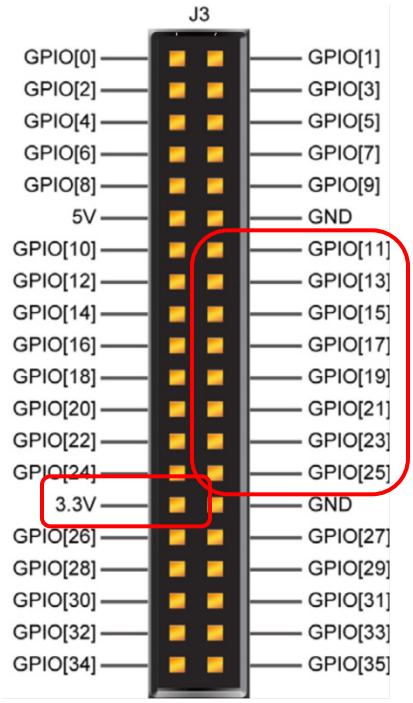

Figure 6. GPIO pinout. The notch is on the left. Image source: Altera

6. Verify that you can wiggle the output pins and read the input pins on the GPIO header.
7. Verify that you know how to build a counter, e.g., to divide the clock.
8. Create a module that can scan the keypad, moving a single 0 across the columns, stopping if any row goes to 0, outputting the character code corresponding to the key that's been pressed. Wire it up to one of the seven segment displays.
9. Add a 16-bit counter that's incremented every time you get a new keystroke.
10. Add a reset function, tied to one of the pushbuttons on the DE1-SoC board.

11. Debug your design, demo it and submit your .v file.

4.1 System Builder

Use the System Builder tool to create an empty KeypadScanner project with the appropriate inputs and outputs as shown in figure 7.

Your finished keyboard scanner will only need the clock, the 7-segment displays and the GPIO-0 header. But you'll likely find it helpful during debug to have the switches and LEDs available as well.

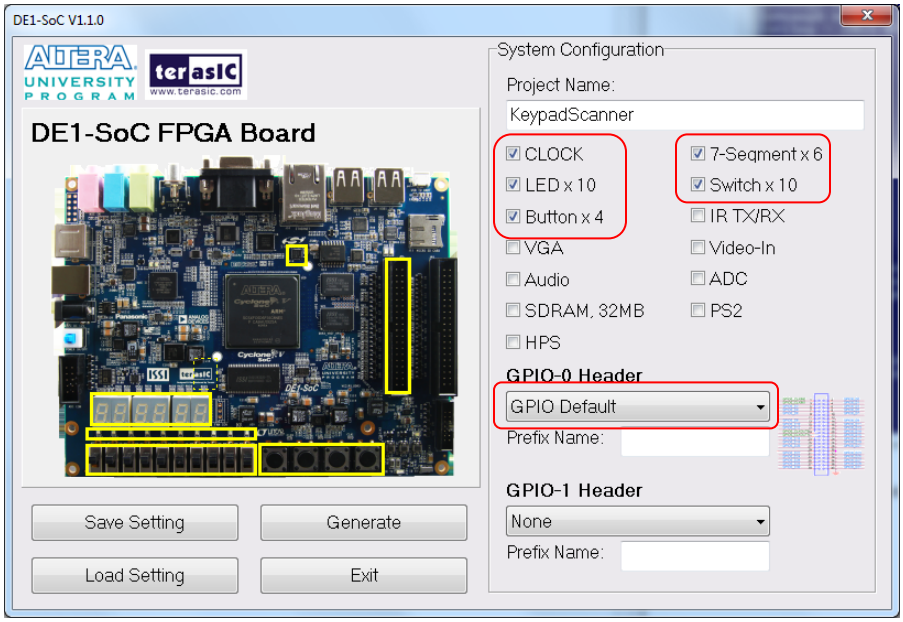

Figure 7. The inputs and outputs you'll need.

4.2 Quartus Prime

Open the project in Quartus Prime, add the KeypadScanner.v source file and open it for editing. Copy and paste your SevenSegment decoder module into your file.

4.3 Verify the GPIO

Devise your own simple experiments using the switches, the LEDs and the multimeter to satisfy yourself that you know how to write to GPIO pins as outputs and read from them as inputs.

There is nothing you need to turn in from this step but please be sure you have this part working as expected before proceeding.

4.4 Clock divider

Verify that you know how use CLOCK_50, the 50 MHz clock, e.g., by building a simple counter that can divide the clock down low enough that you can watch the output change on the LEDs or the 7-segment displays.

Figure 8 shows an example that divides the clock by 10,000, meaning it will go to 0 every 10,000/50 MHz = 200 µs. That's still way too fast for the human eye. But if you made it a 32-bit counter that divided by 100,000,000, it would take 2 s, which would be noticeable. The always block in the example is entered on the positive edge of the clock. (You can choose any clock and either edge but stick with whatever you choose throughout your design.)

This is a clocked, not combinatorial logic. In clocked logic, the intended next state, in this case, the next value of the counter, is calculated based on the current state and then clocked (written) simultaneously into all the outputs at the next clock edge, by which time we expect all the inputs will have settled.

```
module ClockDivider( input CLOCK_50,
      output reg [ 15:0 ] count );

   parameter clockDivisor = 10_000;

   always @( posedge CLOCK_50 )
      if ( count == 0 )
         count <= clockDivisor;
      else
         count <= count - 1;

endmodule
```

Figure 8. A simple clock divider.

Because this is intended as sequential clocked logic where the outputs should all change at once, the assignments use the "<=" operator instead of the "=" operator used for combinatorial logic. The right-hand side of each "<=" assignment is evaluated using the values *at entry* to the always block. *The actual assignment to the variable on the left is deferred until after the end of the always block and is done simultaneously with all the others.*

Never mix combinatorial "=" assignments with clocked "<=" assignments in the same always block. If the always block is clocked sequential logic, use only the "<=" operator. If it's combinatorial, use only the "=" operator.

4.5 Scan the keypad

Create a new module with the following inputs and outputs that simply scans until it finds a key that's pressed, stays there so long as the key remains pressed, and then starts scanning again if the key is released, even for one clock.

```
module Scan( input CLOCK_50, inout [ 7:0 ] keypad,
      output reg [ 3:0 ] rawKey, output reg rawValid );
```

Bits 7:4 of the keypad are the rows. Bits 3:0 are the columns. If a key is being pressed, rawValid should equal 1 and rawKey should equal the *character code* (not the scan code) of that key.

Your module will need to do the following.

1. Create a 2-bit counter as a column number.
2. Decode the column number to drive the selected column to 0 and the rest to z (high impedance).

3. Divide the clock down to a sensible rate for scanning across columns. I chose 100 KHz, meaning I stepped from one column to the next every 500 clocks = 10 µs.
4. Increment the column number every time the clock divisor goes to zero but only if none of the rows is currently 0. (If we've found a key that's pressed, stay on that column.)
5. If a key is pressed, translate the row and column number coordinates into the corresponding hex value as rawKey.

Instantiate a copy of your Scan module and wire it to the keypad and to the rightmost of the 7-segment displays.

4.6 Add a counter

Add a 16-bit counter of the number of key depressions. Each time rawValid goes to a 1, increment the counter. Display it as a hex number in the 4 leftmost 7-segment displays with zero suppression of the three high-order digits. Add a reset function tied to the leftmost button.

What you likely observe is that the keypad works only sometimes. Sometimes when you press a key, the count goes up by only 1 as it should. But it sometimes goes up by 2 or 3 or maybe more because the key is bouncing.

5 Demo and submit

Demo your design and submit your code.