# BEE 271 Digital circuits and systems
# Spring 2017
# Lab 3: Keypad scanner[1]

## 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.

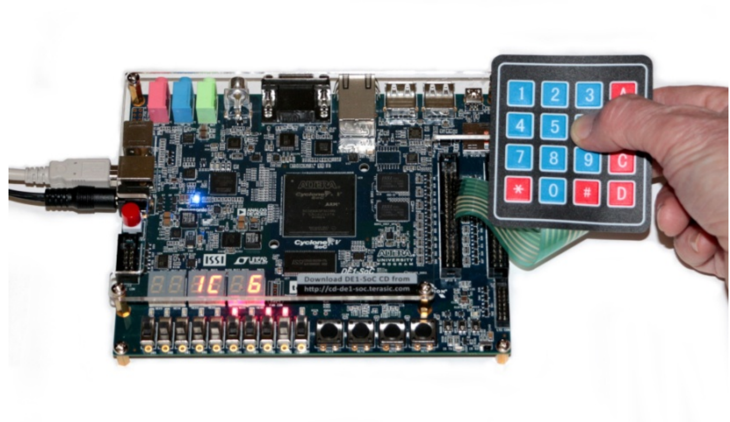

*Figure 1. When a key is pressed, it's displayed and the count is incremented*

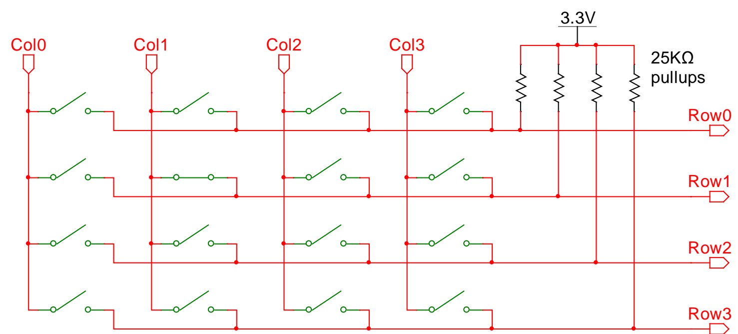


*Figure 2. Keypad with pull-up resistors. The columns are the inputs and the rows are the outputs.*

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.

[1] This lab was written by Nicole Hamilton.

## 2 Work product

At the end of this lab, you must demo your design and submit your code as a .v or .vs 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 the number of keys, $n_k$, not linearly.

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

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 pin has an internal weak pull-up on the FPGA, 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. The on-chip weak pull-ups are 25KΩ; from Ohm's Law we can calculate it will take only 132 µA to pull a row to ground.

$$I = \frac{E}{R} = \frac{3.3\ V}{25\ K\Omega} = 132\ \mu\text{A}$$

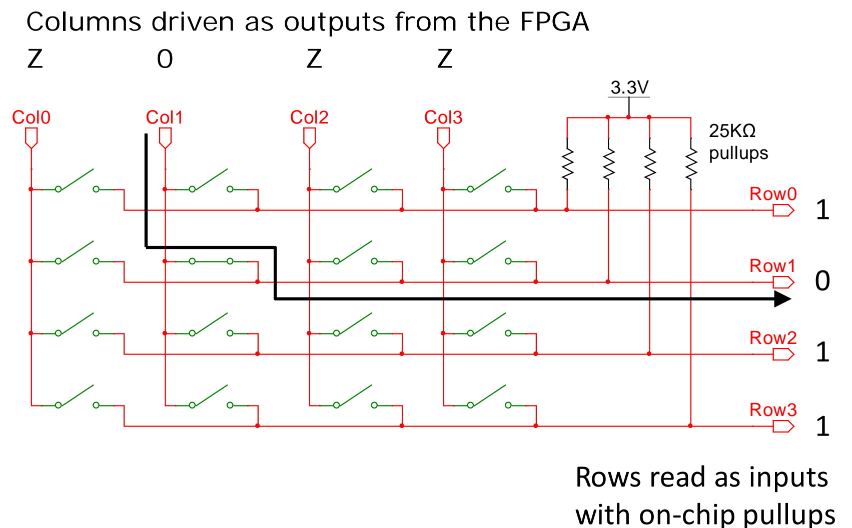


*Figure 3. A closed switch creates a path to pull a row to 0 with only 132 µA.*

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 Z's (high impedance, like it's not connected) 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.

1. Use the System Builder tool and Quartus Prime to create an empty Verilog project with the necessary inputs and outputs.
2. Build a simple counter with the high bits connected to the LEDs and observe the behavior.
3. Convert a two-bit number into a one-hot.
4. Plug the keypad into the GPIO-1 header.
5. Add internal pull-ups to the rows.
6. Add your seven segment decoder module.
7. Modify your counter so that it 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.
8. Add a 16-bit counter that's incremented every time you get a new keystroke.
9. Add a reset function, tied to one of the pushbuttons on the DE1-SoC board.
10. Debug your design, demo it and submit your .v or .sv 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 4.

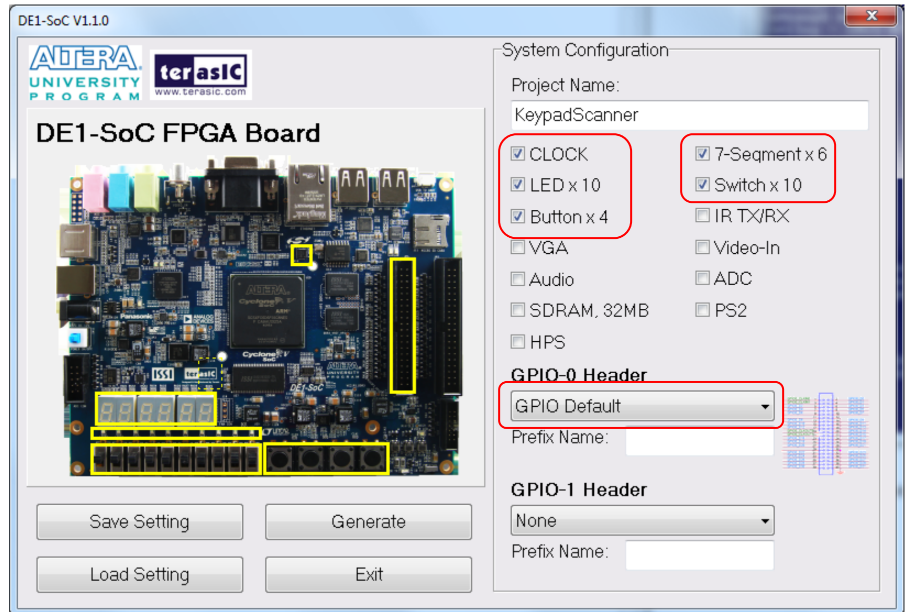


*Figure 4. The inputs and outputs you'll need.*

### 4.2 Quartus Prime

Open the project in Quartus Prime, add the KeypadScanner.v source file, open it for editing and copy your 7-segment decoder module into it.

### 4.3 A simple counter

The first step is verify that you know how use CLOCK_50, the 50 MHz clock by building a simple counter that can divide the clock down low enough that you can watch the output change on the LEDs.

Figure 5 shows a simple module that increments a 32-bit counter with the high-order 10 bits wired to the LEDs.

```
module Scan( input CLOCK_50,
    output [ 9:0 ] LEDR );

  reg [ 31:0 ] counter;

  assign LEDR = counter[ 31:22 ];

  always @( posedge CLOCK_50 )
    counter <= counter + 1;

endmodule
```

*Figure 5. A simple counter to start.*

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 clock edge.

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.

Instantiate a copy of this in your main module and compile and run this on your board. What you should observe is the LEDs counting in binary, each LED blinking at half the frequency of the one to its right.

With a 50 MHz clock, counter[0] will flip from 0 to 1, then back to 0 in two clocks, meaning it will be running at 25 MHz, half the input clock. By extension, each bit k of the counter will have a frequency and period as follows:

$$f_k = \frac{50\ MHz}{2^{k+1}}$$

$$T_k = \frac{1}{f_k}$$

Thus, we'd expect LEDR[0] to blink at about $50e6/2^{23}$ = 6 Hz. LEDR[9] should blink at $50e6/2^{32}$ = .01 Hz (a period of 86 s.)

This should give you an idea of how you might choose two bits from your counter as your column number. If you know the frequency you want, you can calculate which bit will flip at that rate as follows:

$$k = \log_2\left(\frac{50\ MHz}{f_k}\right) - 1$$

## 4.4 Turn that into a one-hot

You then need to turn the two-bit column number into a one-hot as shown in figure 6. Your eventual objective will be to scan from one column to the next at perhaps 50 KHz to 100 KHz. But to make it possible to watch things change on the LEDs, we need something slower, so let's pick fk = 1 Hz, giving k = 25.

In this second step, I've picked counter[ 26:25 ] as the two-bit column number and then translated that to a one-hot output, wiring the column number and the one-hot to the LEDs.

```
module Scan( input CLOCK_50,
      output [ 9:0 ] LEDR );

   reg  [ 31:0 ] counter;
   reg  [  3:0 ] onehot;
   wire [  1:0 ] columnNumber;

   assign columnNumber = counter[ 26:25 ];
   assign LEDR = { onehot, columnNumber };

   always @( posedge CLOCK_50 )
      counter <= counter + 1;

   always @( * )
      case ( columnNumber )
         0: onehot = 'b1000;
         1: onehot = 'b0100;
         2: onehot = 'b0010;
         3: onehot = 'b0001;
      endcase

endmodule
```

*Figure 6. A one-hot counter.*

Compile and run this on your board. What you should observe is 4 LEDs rotating a one hot pattern and 2 LEDs counting 0 to 3 in binary, changing at a 1 Hz rate.

## 4.5 A more sophisticated counter

The previous example did not allow much fine-tuning of the frequency. Figure 7 shows a slightly more sophisticated counter that allows the frequency to be set to any desired value, shown here as 1/3 Hz. Notice the use of floating point in the calculation.

```
module Scan( input CLOCK_50,
      output [ 9:0 ] LEDR );

   reg [ 31:0 ] counter;
   reg [  3:0 ] onehot;
   reg [  1:0 ] columnNumber;

   parameter desiredFrequency = 1.0/3.0,
      divisor = 50_000_000 / desiredFrequency;

   assign LEDR = { onehot, columnNumber };

   always @( posedge CLOCK_50 )
      if ( counter == 0 )
         begin
         counter <= divisor;
         columnNumber <= columnNumber + 1;
         case ( columnNumber )
            0: onehot <= 4'b1000;
            1: onehot <= 4'b0100;
            2: onehot <= 4'b0010;
            3: onehot <= 4'b0001;
         endcase
         end
      else
         counter <= counter - 1;

endmodule
```

*Figure 7. A more sophisticated counter.*

## 4.6 Plug in the keypad

Plug the keypad into the GPIO-1 with the leftmost pin, Row[ 0 ], at GPIO[ 25 ] and the rightmost, Col[ 3 ], at GPIO[ 11 ], as shown in figures 7 and 8.

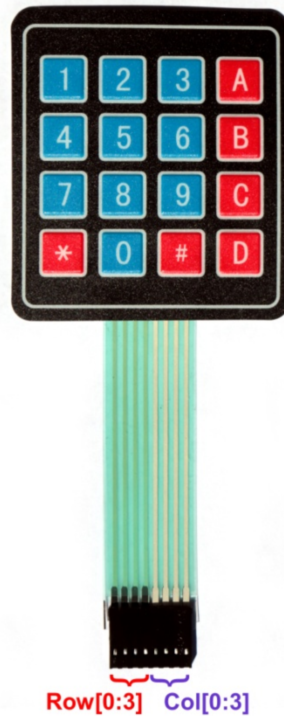


*Figure 7. Keypad pinout.*

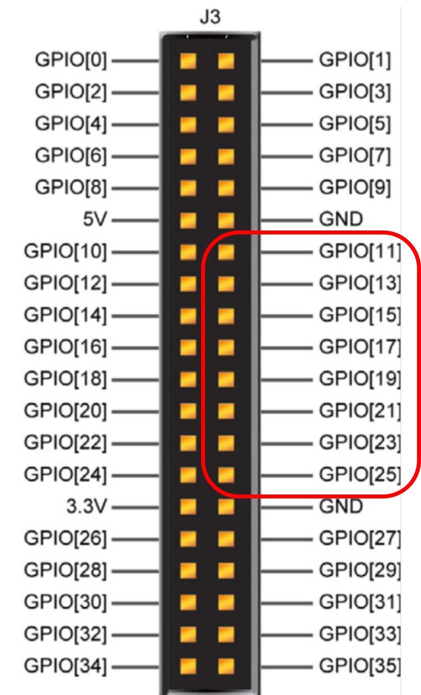


*Figure 8. GPIO pinout. The notch is on the left. Image source: Altera*

## 4.7 Add internal pull-ups

The next step is to enable the internal weak pull-up resistors on the rows. From the main Quartus menu bar, go to Assignments → Assignment Editor.

*Do not change any lines already there* but *add* the last 4 lines shown in figure 9, assigning weak pull-ups to GPIO pins 19, 21, 23 and 25, corresponding to the rows on the keypad.

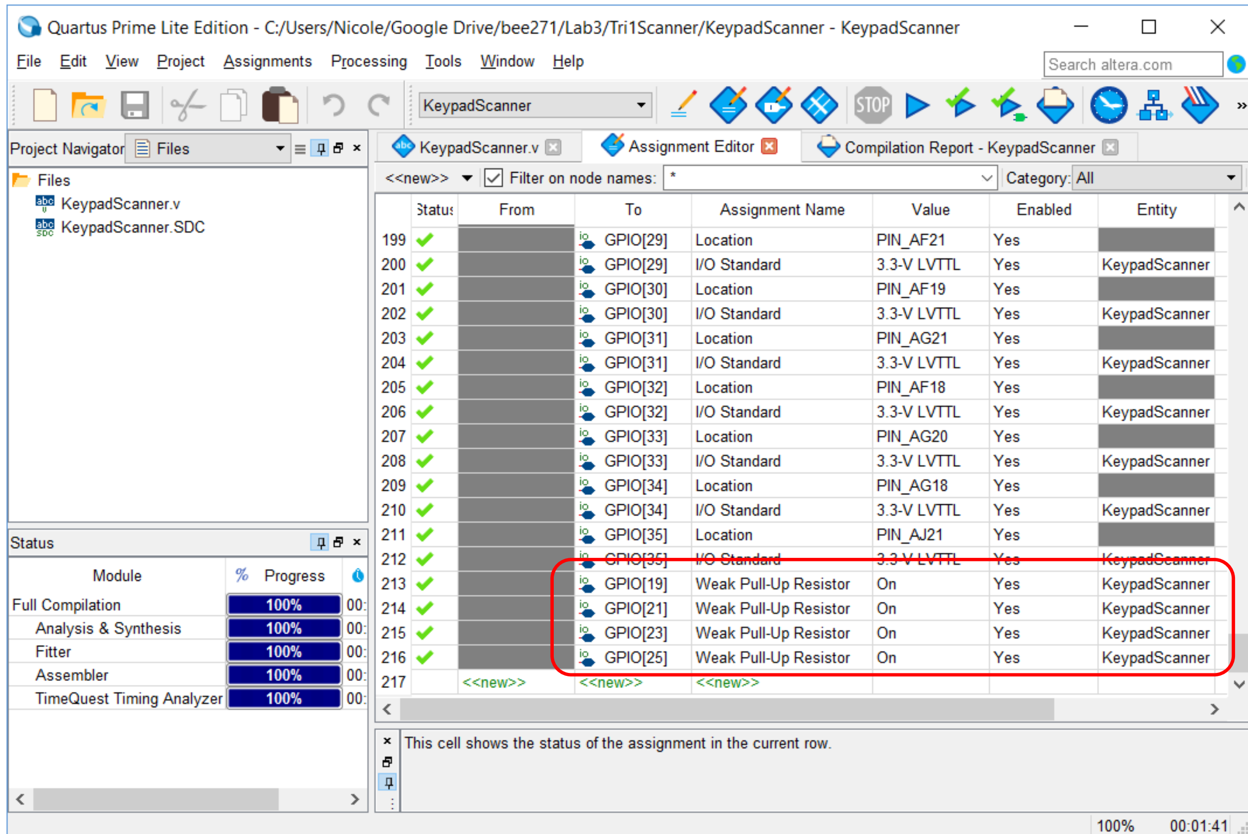


*Figure 9. Adding weak pull-ups.*

The result will be to add the following lines near the bottom of your .qsf (Quartus Settings File) file. (If you prefer, you can simply edit the .qsf file directly to add these lines.)

```
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to GPIO[19]
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to GPIO[21]
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to GPIO[23]
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to GPIO[25]
```

## 4.8 Scan the keypad

Modify your Scan module to have the following inputs and outputs.

```
    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* (hex 0 through F) of that key.

To hook this up to the GPIO pins, the instantiation in your main module should look something like this. Notice how concatenation is used to collect the GPIO pins connected to the keypad into an 8-bit vector to be passed to the scan module.

```
wire [ 3:0 ] rawKey;
wire rawValid;

Scan sc( CLOCK_50,
   { GPIO[ 25 ], GPIO[ 23 ], GPIO[ 21 ], GPIO[ 19 ],
   GPIO[ 17 ], GPIO[ 15 ], GPIO[ 13 ], GPIO[ 11 ] },
   rawKey, rawValid );
```

Modify the scan module so that 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.

1. Modify your one-hot code to drive the selected column to 0 and the rest to z (high impedance) rather than 1.
2. Pick a sensible rate for scanning across columns, somewhere between 50 and 100 KHz.
3. Increment the column number only if none of the rows is currently 0. (If we've found a key that's pressed, stay on that column.)
4. If a key is pressed, translate the row and column number coordinates into the corresponding hex value as rawKey.
5. Wire the rawKey to the rightmost of the 7-segment displays (HEX0) but only display the digit if rawValid = 1.

### 4.9 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.