Lab 6: Combination Lock & Keypad, Part 2

You must work on this lab <u>with a partner</u>. V3.

Objective

The purpose of this lab is to learn to design more complicated FSMs to drive an external logic circuit. You will use behavioural SystemVerilog design with Quartus.

Similar to Lab 5, design a circuit that locks and unlocks a safe that you might find in a hotel room. In this lab, however, you will use a numeric keypad to enter the combination.

- To lock the safe, enter any combination as a 6-digit password and press #
- To unlock the safe, enter the same password and press #
- When entering the password, press * to cancel and start over
- When ready to start accepting a password, display `_OPEn_` or `LOCHED`, as appropriate
- For visual confirmation, incrementally display the password as each key is pressed

1. Equipment

You will need:

- Two FPGA boards (DE10-Lite and DE1-SoC) and two computers
- One 16-key (4x4) numeric keypad with keys labelled 0 to 9, A to D, # (aka E) and * (aka F)
- Prebuilt jumper wires (the rainbow cable) with male (M) and female (F) ends:
    - 8 + 6 MM (two DE10-Lite boards), 6 FF + 8 MF (two DE1-SoC boards), and 8 MM/MF + 6 MF/MM (mixed boards; the `combo` one uses 6 wires),
    - *Note: **<u>two DE1-SoC boards</u>** require FF cables that are **<u>NOT included</u>** in your Parts kit. You can get creative, e.g. connect the male ends of two MF cables with your breadboard.*

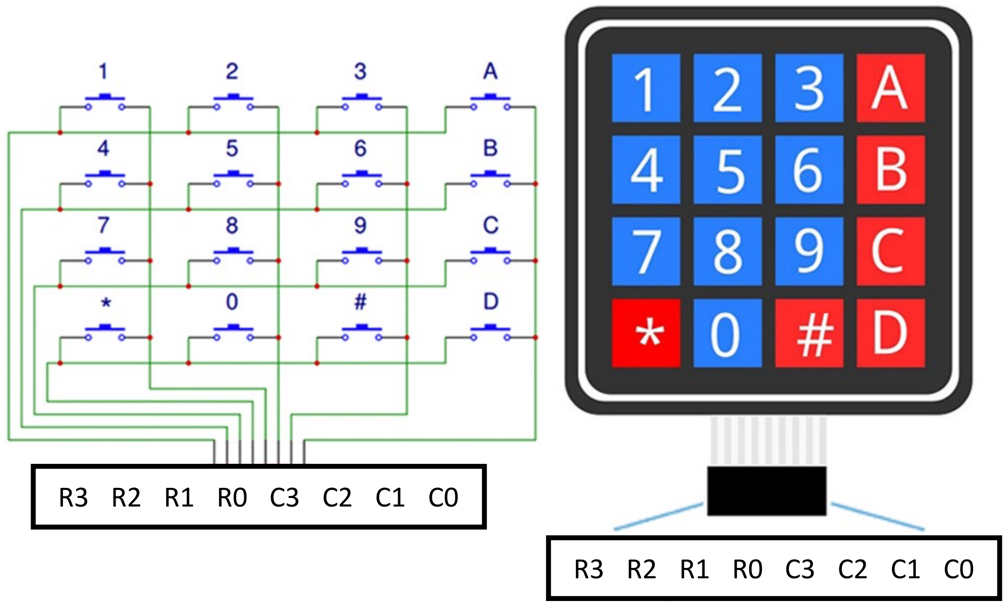

Figure 1. Keypad schematic.

2. Overall Design

This lab will be implemented as two separate designs on two connected FPGA boards:

- `keyboard` accepts user input, debounces the pressed key, and both displays/transmits it
- `combo` receives a key from `keyboard` and displays the password entry, _OPEn_ or LOCHED
- students without a partner will design the `keyboard` module to use only one FPGA board

A precompiled **golden reference solution** for each module is available as a bitstream (.SOF file) for both DE10-Lite and DE1-SoC boards, allowing you to test against a known working solution for each design.

Start early – this lab will be more challenging than previous labs. If you fail to get one module working, you may use the reference solution to demonstrate that your other module works properly.

The most difficult part of this lab is handling metastability and keypad switch bounce, so it is provided to you on Canvas as a SystemVerilog module `kb_db`.

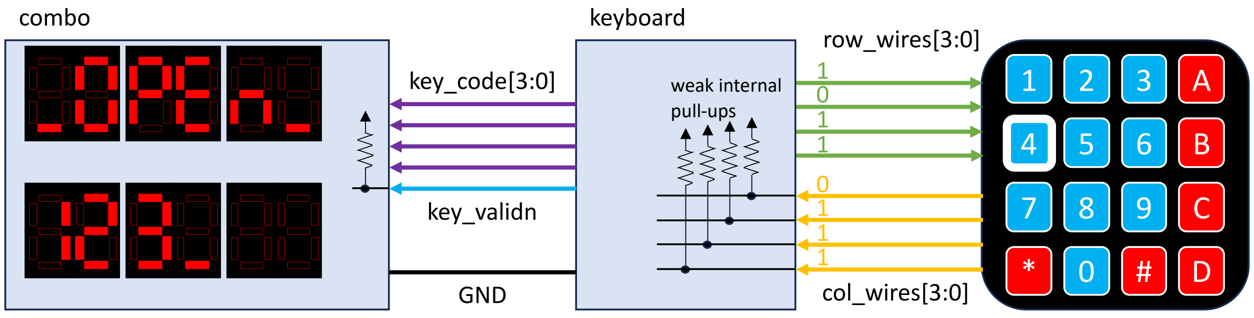

Figure 2. Block diagram showing board-to-board and board-to-keyboard connections.

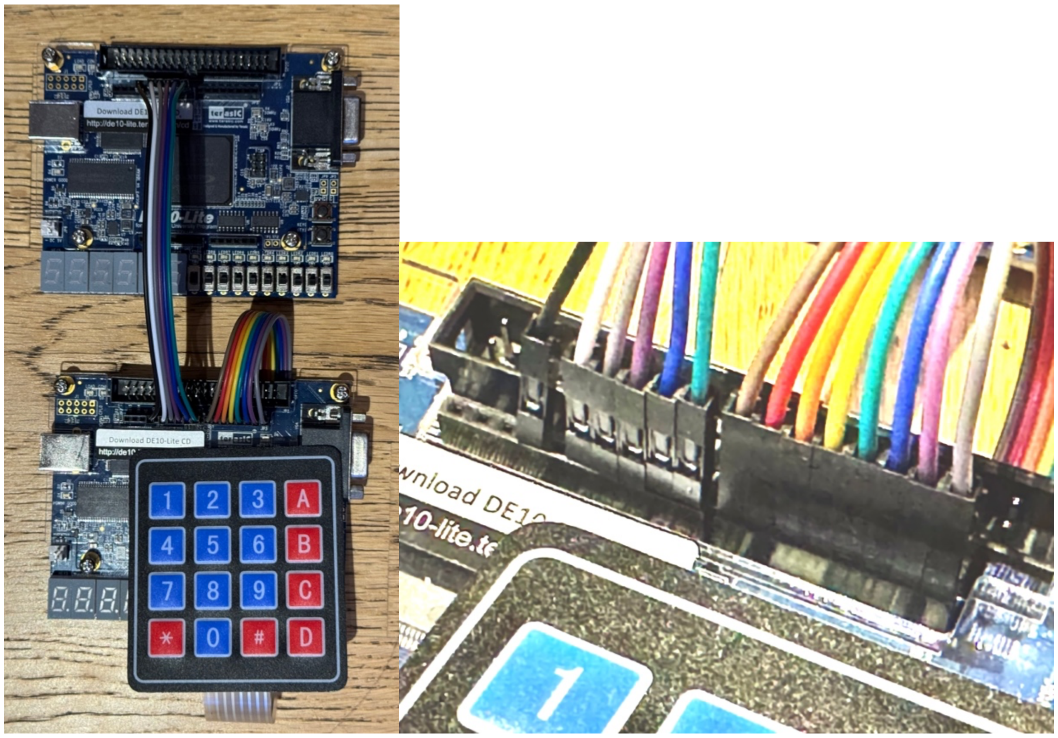

Figure 3. DE10-Lite combination lock (combo board on top) with wiring close-up.

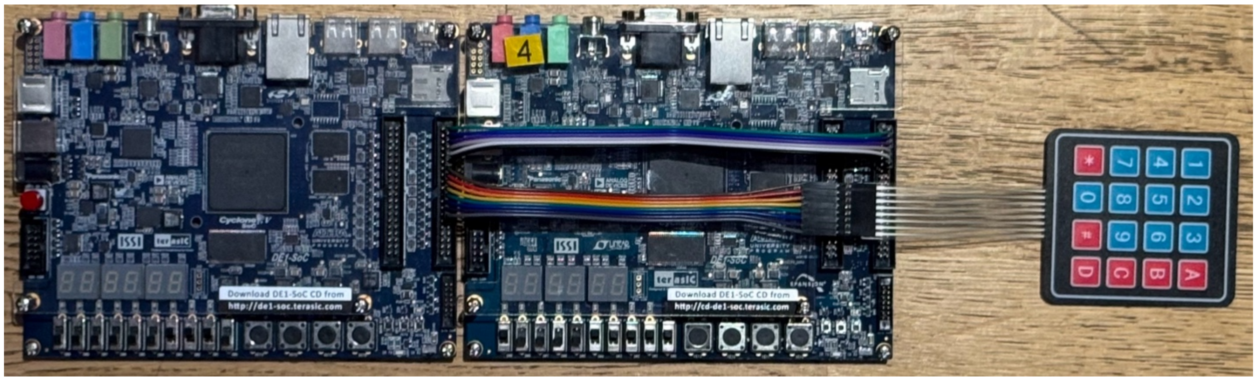
Figure 4. DE1-SoC combination lock (`combo` board on right).

3. Approach

Begin by testing your keypad. Connect it to your FPGA board according to Figure 5 and use the golden reference solution `keyboard.sof` on Canvas. If there is a problem, get a replacement.

Build and test your solution incrementally:

1. Test your equipment
    - connect your boards and keypad together and program the two golden reference solutions
    - test the keypad and wiring to the golden `keyboard` module
    - test the wiring to the golden `combo` module
2. Build and test your SystemVerilog modules independently
    - `combo` module tips
        - start using simulation and a testbench: you can also print Verilog debug messages with statement `$display("_OPEn_"); or $display("LOCHED");`
        - start with just a 1-digit or 2-digit password
        - single-board testing: use `SW[3:0]` to simulate `key_code[3:0]` and `KEY[1]` to simulate `key_validn`
            - the golden reference accepts these manual inputs as well
            - the golden reference displays the last `key_code[3:0]` on `LEDR[9:6]`
        - `key_validn` is active low and indicates when the values on `key_code[3:0]` are valid; during this time, `key_code[3:0]` will not change
        - multi-board testing: connect to the golden reference `keyboard` solution
    - `keyboard` module tips
        - start using simulation and a testbench
        - use the 7-segment display to display the last 6 key presses
        - single-board testing: use `SW[7:0]` to provide `{row, col}` input values, and `KEY[1]` to simulate a keypress
            - the golden reference does not accept these inputs
            - referring to Figure 3, the golden reference displays:
                - `LEDR[9]=valid` and `LEDR[8]=debouncedOK`
                - `LEDR[7:4]=~row` and `LEDR[3:0]=~col`
        - multi-board testing: connect to the golden reference `combo` solution
3. Finally, connect your two modules together!

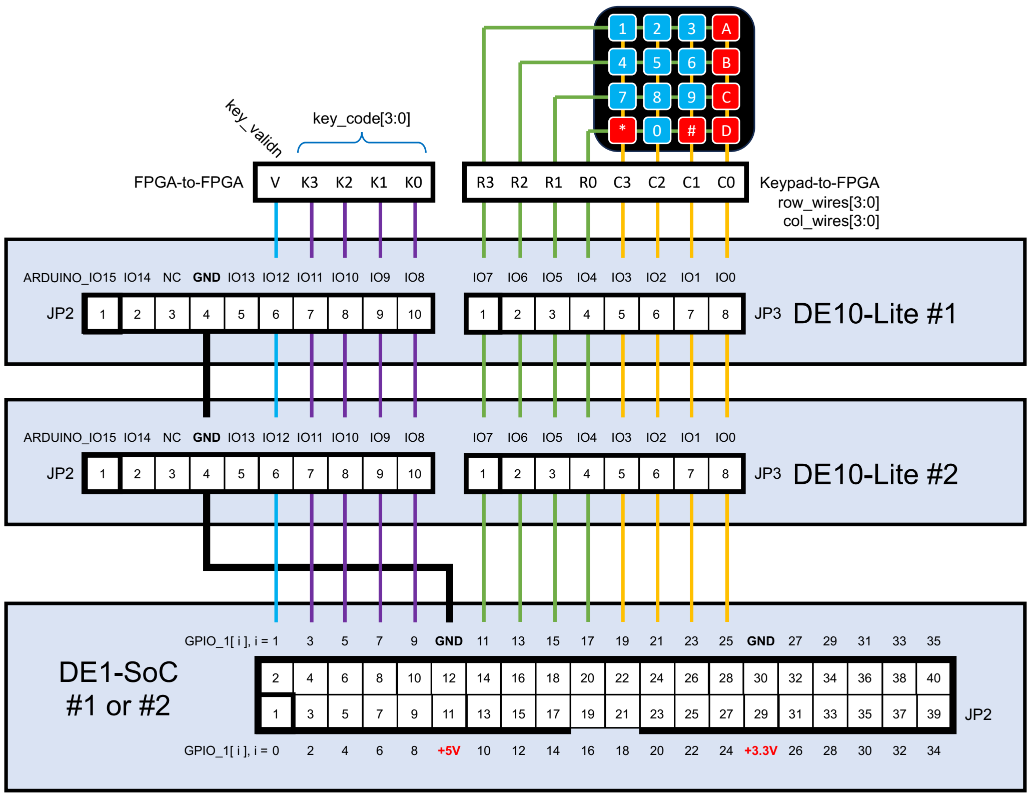

Figure 5. Connector pinouts.

4. Inout Wires and GPIO Connections

The connections between FPGA boards, and from FPGA board to keypad, are shown in Figure 2. We will be using some of the **general-purpose input/output (gpio or I/O) pins** on the FPGA boards. Each pin can be configured to be either an input pin, or an output pin, or both (bidirectional).

To avoid damaging your FPGA, NEVER connect two OUTPUT pins together. In particular, it is easy to accidentally configure both FPGAs as `keyboard` and thereby short the `key_code[3:0]` and `key_validn` outputs between two boards.

On the DE10-Lite, we will be using the two Arduino connectors labelled J2 and J3 on the PCB.

- These female connectors accept a male (M) wire end.
- In SystemVerilog, these signals are named `ARDUINO_IO[15:0]`. See Figure 5.

On the DE1-SoC, we will be using the GPIO_1 connector also labelled JP2 on the PCB.

- These are called ‘header connectors’ with male pins, so they accept a female (F) wire end.
- In SystemVerilog, these signals are named `GPIO_1[35:0]`**.** See Figure 5.

For both DE10-Lite and DE1-SoC connectors, the location of connector pin 1 can be confirmed on the bottom side of the circuit board by the square solder joint. The DE1-SoC also has a cutout in the plastic shroud on the top side next to pins 19 and 21.

In SystemVerilog, connections to these general-purpose I/O signals should be declared as type ‘`inout wire`’. Using ‘`input logic`’ or ‘`output logic`’ also works when all of the pins are the same direction (ie, all inputs or all outputs). However, the ‘`inout wire`’ declaration allows you to set the direction of each pin separately, including using the pin as a bidirectional pin (using a tri-state gate); inside your logic module, use a continuous assignment statement to clearly indicate your intent, e.g.:

```
logic [3:0] row_wires, col_wires; // use 'wire' or 'logic'
assign ARDUINO_IO[7:4] = row_wires; // create gpio outputs
assign col_wires = ARDUINO_IO[3:0]; // create gpio inputs
```

5. Combo Module

The `combo` module implements the FSM that allows the user to operate the safe as described in the Objective section. It uses the 7-segment display to show the current state of the FSM, including open, unlocked, or a partially entered password.

The inputs to the FSM come from the `keyboard` module implemented on a separate FPGA board, giving rise to a few important technical issues:

- You **must connect the Ground (0V)** between boards so they have a common reference voltage. Without this, the receiving board will not know the proper voltage being supplied by the other board, so it may consider the input to be in the forbidden zone or misinterpret the 0 or 1 value.
- **Do NOT connect +5V or +3.3V or any other power signals.** The two boards have independent power supplies; any small differences in their voltage would cause the two power supplies to draw a large current from one with the higher voltage.
- Although both boards have highly accurate 50MHz clock, these clocks operate independently and will not perfectly matched; one is likely to be a bit faster than the others, and the edges will never be perfectly coincident (called phase alignment). For reliable communications, we will practice some simple rules:
    - To avoid metastability, the receiver must use a synchronizer. This means each of the 5 signals (`key_validn` and `key_code[3:0]`) must pass through two flip-flops in series, all clocked by the receiver's clock. Call these synchronized signals `key_validn_sync` and `key_code_sync[3:0]`.
    - The transmitter must hold the `key_validn` and `key_code[3:0]` signals constant for at least 3 cycles so the receiver has time to recognize the inputs even if its clock is faster. The `key_validn` signal will only be asserted (low) when `key_code[3:0]` is sending the correct value.
    - After the synchronizer, the receiver must sample `key_code_sync[3:0]` one cycle after it sees `key_validn_sync` go low.
        - Because of metastability, any of these 5 signals may arrive on one clock cycle, and the remainder will appear on the next cycle.
        - Therefore, sampling a set of parallel signals *on the same cycle* its valid signal is asserted could lead to incorrect results; if any of the signals (including the valid signal) is delayed (or advanced) by a small amount and/or fails to meet the setup time of its flip-flop, its value may appear one cycle early, or it may one cycle later, than the other signals.
        - Ensuring all signals are constant (stable) for 3 cycles, and capturing the `key_code_sync[3:0]` values in the middle of the time period while `key_validn_sync` is low, is a simple and safe way to ensure success.
        - There are many other ways this could be done in practice, including: source-synchronous transmission, phase-locking clocks with PLLs, distributing globally phase-aligned clocks, sending data and clock and recovering them on the other end through clock-data recovery, and asynchronous transmission over a serial wire (see UART in Wikipedia). Don't attempt any of these!

Add a weak pull-up resistor to the `key_validn` GPIO pin:

- The pullup resistor ensures the `combo` module sees a logic 1, rather than a floating input value, when the `keyboard` FPGA isn't attached.
- Quartus → Assignments → Assignment Editor → scroll to bottom → in the "To" column, enter a pin name (e.g., `ARDUINO_IO[12]`) where it says <<new>> → in the "Assignment Name" column, double-click and select "Weak Pull-Up Resistor" near the bottom →in the "Value" column, double-click and select "On" → in the "Enabled" column, make sure it says "Yes".
    - For DE1-SoC, use `GPIO_1[1]` instead.
    - Optionally, you can also add pullups to the `key_code[3:0]` pins.

6. Keyboard Module

The `keyboard` module continuously scans the keypad and transmits each key press to another module. It is governed by an FSM that must take into account a few technical issues:

- Metastability: the user can press a key at any time relative to the clock.
- Bounce: The key may bounce when pressed; use a timer to detect when it is stable.
- Holding a key down should result in a single keypress, not several keypresses.
- Use hexadecimal values for `key_code[3:0]`. Use 4'hE and 4'hF for # and *, respectively.
- `key_code[3:0]` and `key_validn` must be held constant the entire time a key is pressed, allowing it to also detect when the key is released. ~~for at least 3 cycles for each keypress.~~

We have prewritten a module called `kb_db` to perform debouncing and resolve metastability (the first two issues). However, your FSM must resolve all of the other issues.

The schematic and SystemVerilog code for `kb_db` are presented in Figures 6 and 7, respectively. The `valid` output is asserted (high) only while the user holds down a key and it is safe to inspect the `row[3:0` and `col[3:0]` values. It is deasserted by: (a) releasing a key, or (b) changing the incoming `row_scan[3:0]` (driven by your FSM), or (c) any key bounce. You should not need it, but the `debounceOK` is asserted one cycle before `valid` is asserted; it is 0 while keypad results are unreliable.

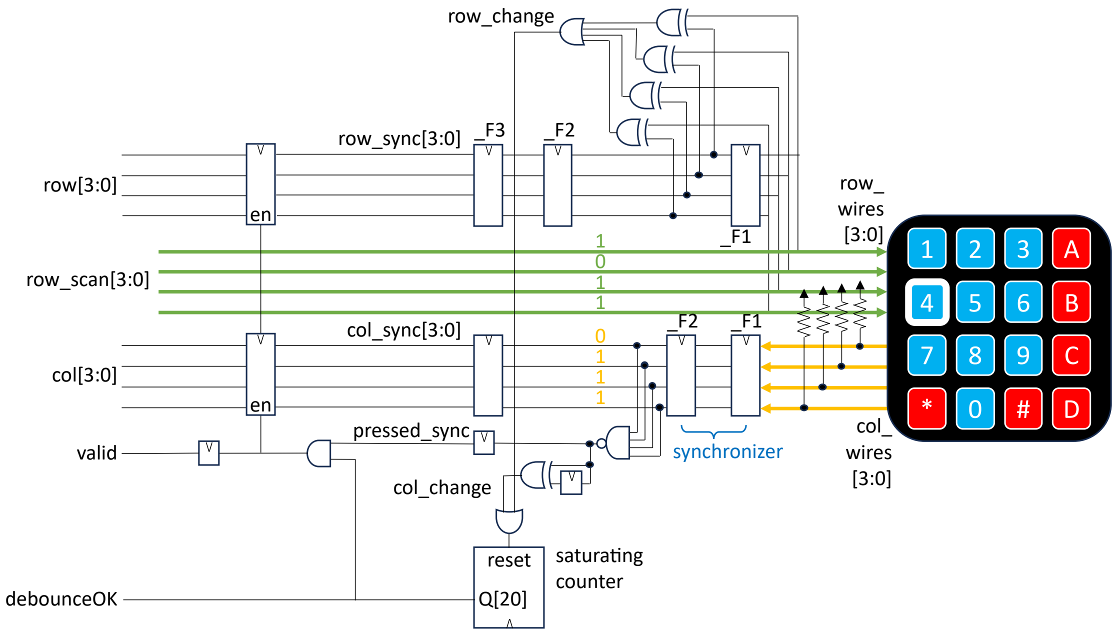

Figure 6. Keypad synchronizer and debounce schematic (see `key_db` module in Figure 5).

```
module kb_db #( DELAY=16 ) (
       input logic         clk,
       input logic         rst,
       inout wire   [3:0]  row_wires, // could be ‘output logic’
       inout wire   [3:0]  col_wires, // could be ‘input logic’
       input logic  [3:0]  row_scan,
       output logic [3:0]  row,
       output logic [3:0]  col,
       output logic        valid,
       output logic        debounceOK
);
       logic [3:0] col_F1, col_F2;
       logic [3:0] row_F1, row_F2;
       logic pressed, row_change, col_change;

       assign row_wires    = row_scan;
       assign pressed      = ~&( col_F2 );
       assign col_change   = pressed ^ pressed_sync;
       assign row_change   = |(row_scan ^ row_F1);

       logic [3:0] row_sync, col_sync;
       logic       pressed_sync;

       // synchronizer
       always_ff @( posedge clk ) begin
              row_F1   <= row_scan;              col_F1   <= col_wires;
              row_F2   <= row_F1;                col_F2   <= col_F1;
              row_sync <= row_F2;                col_sync <= col_F2;
              //
              pressed_sync <= pressed;
       end

       // final retiming flip-flops
       // ensure row/col/valid appear together at the same time
       always_ff @( posedge clk ) begin
              valid <= debounceOK & pressed_sync;
              if( debounceOK & pressed_sync ) begin
                     row   <= row_sync;
                     col   <= col_sync;
              end else begin
                     row   <= 0;
                     col   <= 0;
              end
       end

       // debounce counter
       logic [DELAY:0] counter;
       initial counter = 0;
       always_ff @( posedge clk ) begin
              if( rst | row_change | col_change ) begin
                     counter <= 0;
              end else if( !debounceOK ) begin
                     counter <= counter+1;
              end
       end
       assign debounceOK = counter[DELAY];
endmodule
```

Figure 7. Keypad logic to handle debouncing and metastability.

To decode which key is pressed on the keypad, note:

- As shown in Figure 1, a keypad is incredibly simple. Pressing a specific key results in a simple connection between one specific row wire and one specific column wire. For the 4x4 keypad, there are 4 row wires and 4 column wires; an FSM must determine which key is pressed and transmit the corresponding 4-bit value on `key_code[3:0]`.
- When no key is pressed, the 4 column wires are floating. To make them appear as a logic 1 instead, connect all 4 column wires to Vdd through a pull-up resistor.
    - Instead of wiring an external resistor, you can enable a weak internal pull-up resistor that is available on the FPGA input pins. These resistors are in the range of 50 to 500 kOhm.
- When a key is pressed, the keypad connects the key's column wire to its row wire.
    - To determine which column, look for the 0 on the four column wires.
    - To determine which row, you need to scan row-by-row with an FSM:
        - Output a 1 to all row wires except for one row being activated; it should have a 0.
        - Pressing any key on that row will send a specific column wire to 0; all other column wires remain 1 due to the pull-up resistor.
        - Changing which row is activated, or pressing any key, restarts a debounce counter. When the counter reaches its limit, the `debounceOK` signal is asserted and the column wires are safe to examine.
        - Do not change the activated row too quickly or the debounce counter will be reset before you can examine the column wires. It is a good idea to assert the same row wire for 8 to 16 times longer than the debounce period.

Add 4 weak pull-up resistors to the column GPIO pins:

- Quartus → Assignments → Assignment Editor → scroll to bottom → in the "To" column, enter a pin name (e.g., `ARDUINO_IO[0]`) where it says <<new>> → in the "Assignment Name" column, double-click and select "Weak Pull-Up Resistor" near the bottom →in the "Value" column, double-click and select "On" → in the "Enabled" column, make sure it says "Yes".
- Repeat for each of the other 3 column inputs (e.g., `ARDUINO_IO[1]`, `ARDUINO_IO[2]`, and `ARDUINO_IO[3]`).
    - For DE1-SoC, use `GPIO_1[25]`, `GPIO_1[23]`, `GPIO_1[21]`, and `GPIO_1[19]`.

7. Putting it All Together

When you think your combo and keyboard modules are both 100% correct (and each works with the corresponding golden reference solution), you are ready to try the whole thing. Good luck!

8. What to Turn In and Demonstrate

TBD.