STARS 2024

Lab Experiment 7: Design and Verification of Sequential Logic in SystemVerilog

---

Objectives

- To learn how to convert various sequential circuit designs into Verilog.
- To evaluate these modules on the physical FPGA development board.
- To learn new Verilog/SV syntax for designing and verifying sequential modules.

Work with a partner for this lab!

First, download the template repository for this lab from:
https://github.com/STARS-Design-Track-2024/Lab7_Template

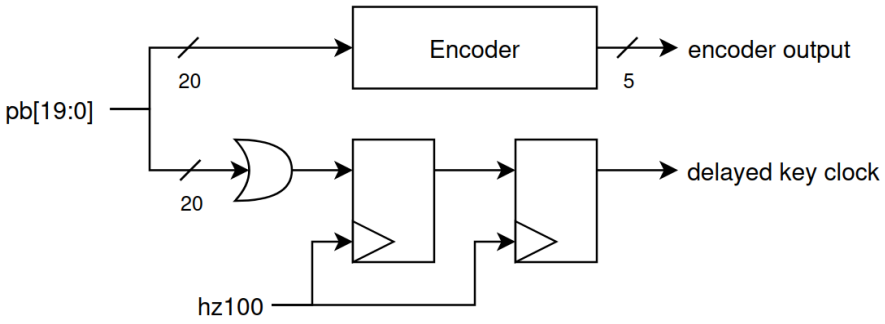
Step 1: Implement and verify your encoder with synchronized strobe

For this step, you should have finished the encoder from the last lab. We can use that encoder to create designs that check the combinational encoder output **out** to know what button was pressed, using the strobe as a clock and therefore as a way of knowing if a button was pressed in the first place.

Copy that same encoder, change the name to **synckey**, and add:

1. Clock and reset ports that should be connected to **hz100** and **reset** in the top module.
2. 2 flip-flops for the synchronizer that use the newly added clock and reset signals, and introduce a delay in the strobe output by passing the ORed value of the button inputs in as data, and connecting the output of the second flip-flop as your **strobe** output.

Here's the block diagram from lecture for reference:



1.1 Implementation

Therefore, your **synckey** module ports are as follows:

Port	Type	Length
clk	input	1
rst	input	1
in	input	20
out	output	5
strobe	output	1

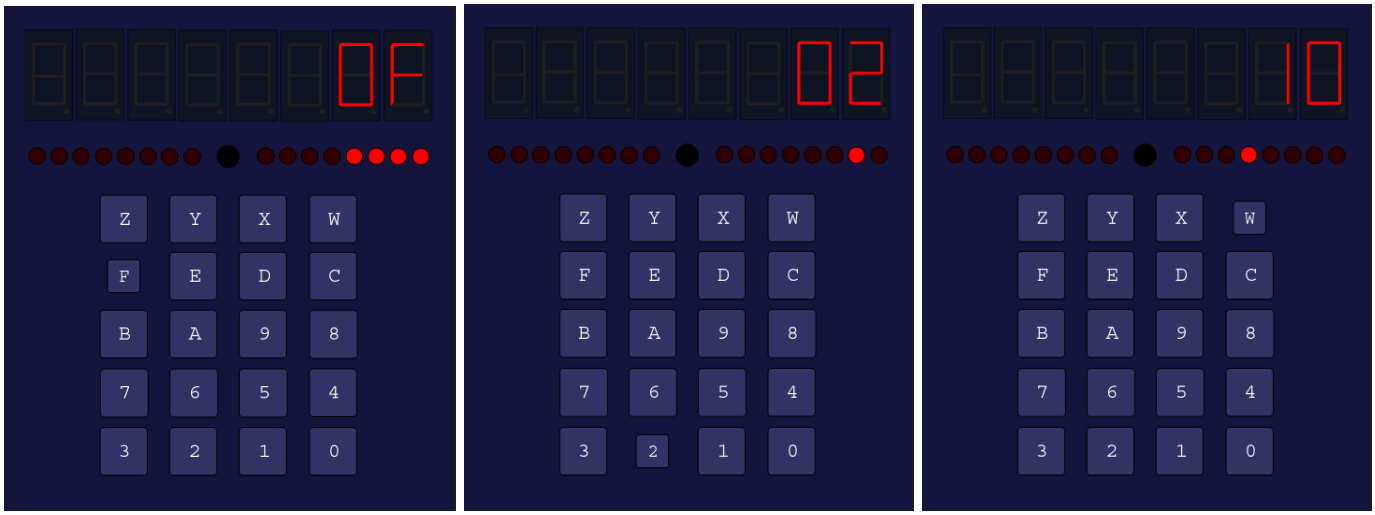
To know if it's working as intended, instantiate the module under **top** with **hz100** as the clock and **reset** as the module reset (which can be pulled high if we press 3, 0 and W), and do the following:

1. Create a 8-bit register that triggers on the rising edge of the **synckey strobe** output clock, and its data as the **synckey** encoder **out**put. (The data is only 5 bits long, so how do you assign it to an 8-bit bus?) The asynchronous reset should be the **top** module **reset**, and the register should reset to 0 on the rising edge of **reset**.
2. Connect the bottom 4 bits as the input to one new **ssdec** instance so that you can see the number on **ss0**.
3. Connect the top 4 bits as the input to a second new **ssdec** instance to see 1 or 0 on **ss1**. (Do you see why we made the register 3 bits longer?)
4. Don't forget to set the enable for both instances to 1!

This should let you see the hexadecimal value of the button you pressed last. For example, pressing and releasing 3 should display "03". Since we created a register that updates only on the rising edge of **strobe**, releasing the button should not change the value back to 0!

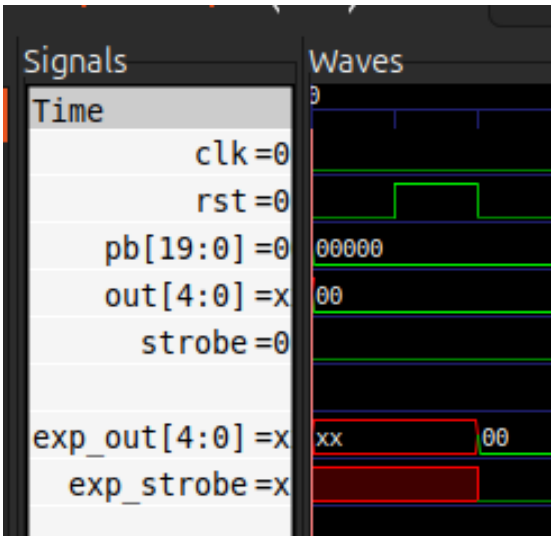
Try this with a few other buttons. Don't be surprised when you see A-F - after all, **ssdec** handles hexadecimal values from 0-F. Another example - pressing Z, which is 19, or 5'b10011, will be represented as "13" on the displays (0001 0011).

Some other examples below (the value of **out** is also connected to **right[4:0]** to let you see the binary representation):



1.2 Verification

Writing a testbench for a sequential design requires that you take into account the clock and reset signals. Refer to the lecture slides to learn how to toggle those in a testbench (automatic toggling for the clock, and using a task to toggle reset).



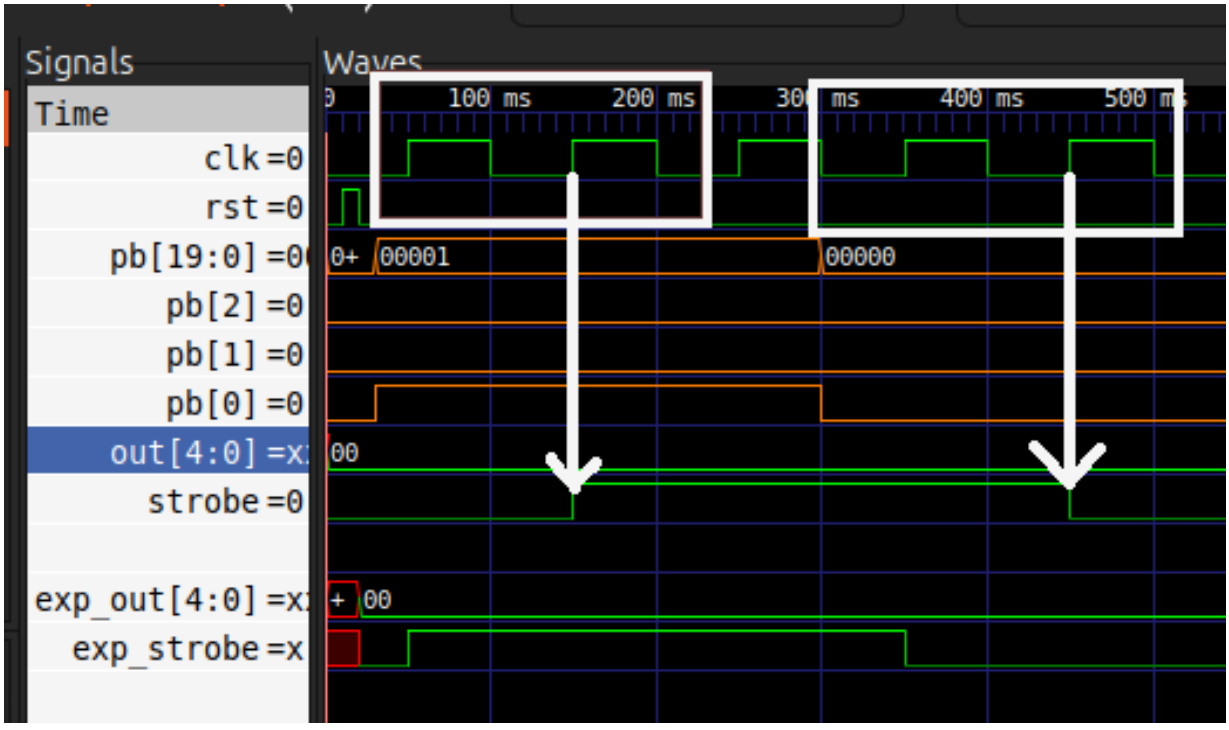
As a reminder, running "make sim" compiles the testbench in **tb.sv** and your module in **top.sv**. If you have multiple testbenches in **tb.sv**, comment them out. However, you should change this to run "make verify_synckey" as explained in the slides.

The bare minimum tests that a sequential design testbench should run are as follows:

- The **power-on reset** case, where the simulation has just started, and you toggle the reset from 0 to 1, and back to 0 using the task you created. Check that the outputs are 0 (or in the case of any other design, the value you intended).

  As shown on the prior page, it doesn't need to be a very long test case.

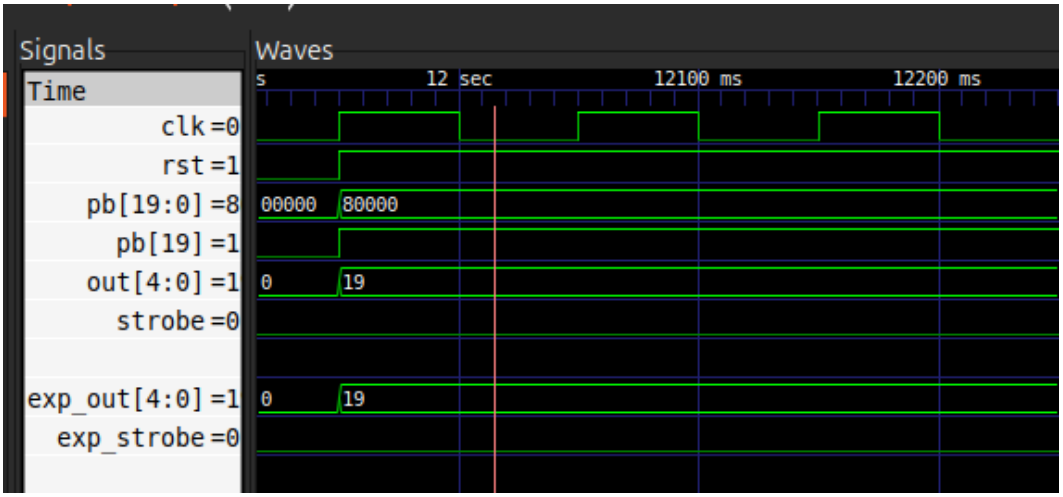
- The **normal operation** cases, where, for each push button, you should:
  - Set **in** to a value that indicates a push button was pressed.
  - After a small wait, test the encoder output **out** and ensure it is <u>not</u> dependent on the clock. **out** changes immediately after a small wait (before the rising edge of the clock arrives - keep track of the time that passes to ensure it does this!), and not on the rising edge of the clock. This is the **combinational** part of the design.
  - While **in** is set to a particular value, wait for **2** rising edges of the clock, and then check that the **strobe** output is now high.
  - Set **in** back to 0, check that **out** changes back to 0, and that **strobe** goes back to 0 2 clock cycles (clock cycle = time to go from one rising edge to another) later.



In this example, we "press" button 0 a little before the rising edge, wait two rising edges of **clk**, and check that **strobe** is now equal to our expected value (which we set back before the

first edge). Then we "release" the button, and wait for **strobe** to clear back to 0. (Why can't we test the next button directly? Consider this - are we properly testing **strobe** for each button if we do that?)

- The **mid-operation reset** case, where a pushbutton is pressed, but then we assert reset. We keep a button pressed to ensure that **out** does not change, since it is not clocked, and that the **strobe** turns off**,** since it's the output of an asynchronously reset flip-flop. This test ensures that your design is capable of resetting properly, even during regular operation.



Common issues while writing sequential testbenches:

- Remember that all variables - test count variables, signals/buses connected to the DUT, etc. **must be initialized to some value**! Do not assume anything is automatically initialized to zero in a testbench. You'll have made this mistake if your testbench output shows "x" (unknown/metastable) instead of 1 or 0.
- **Good timing is essential**. You should not have to worry about cases where the input changes too close to the rising edge, because we already know that would cause metastability. Make sure your inputs change at least 1 millisecond (if you're using a 1ms timescale) before the rising edge arrives. Another good option is changing the inputs at the falling edge.

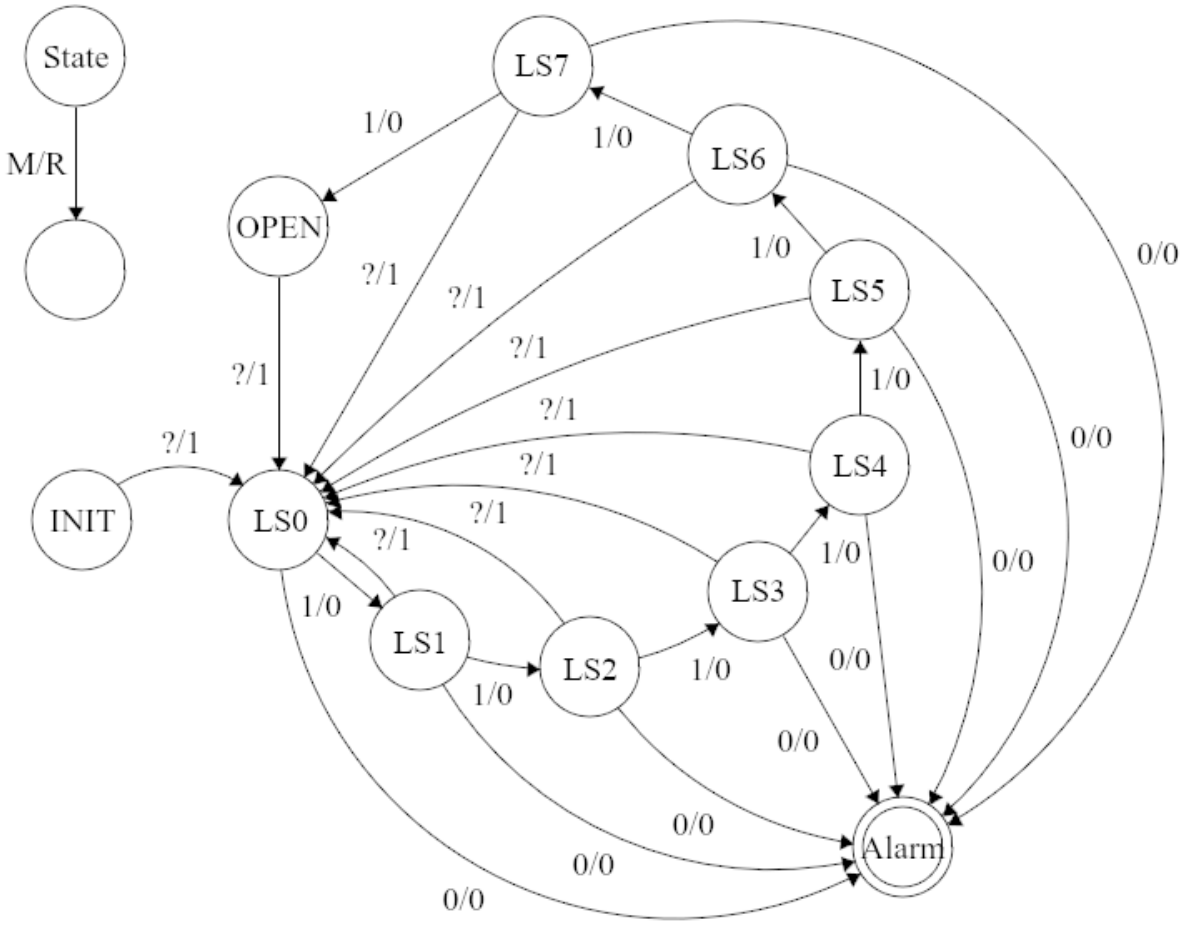
Checkpoint L7C1: **Demonstrate** your new **synckey** module and the testbench you implemented to verify it **to your peer mentor/instructor**. Show that the testbench works for your partner's **synckey** and vice-versa. If your testbench waveforms differ from the ones above, explain what each part does and identify the three basic tests described above with your waveforms. **Make sure to document everything in your lab journal.**

2: Implement and verify a finite state machine

We're going to use our **synckey** as an input mechanism to a **digital lock**. In this step, you'll create the **state machine** for the lock.

The digital lock will have 10 states - a single INIT state where you will enter the 8 digits that will form the **combination** for your lock, 8 VERIFY states where you need to re-enter the same sequence, an OPEN state after all 8 correct digits have been entered, and an ALARM state if a mistake is made.

A sample **state transition diagram** for our lock is as follows:



The LS0-LS7 states are the VERIFY states. M stands for Matched, as in the button being pressed matches the current digit in the sequence. R stands for Relock/ConfiRm, which indicates the "submit" button was pressed (we'll make this W, so **keyout** should be 16).

The digits that can be used as the lock combination are 0 through F.

2.1 Implementation

The **fsm** module ports are as follows:

Port	Type	Length
clk	input	1
rst	input	1
keyout	input	5
seq	input	32 (8 digits x 4 bits)
state	output	4

Since this state machine only changes every time we press a button (verifying the button press against the current sequence), the clock is **strobe** from **synckey**.

rst should be connected to the top-level **reset**. When the module is reset, the lock state should be returned to INIT.

keyout is connected to **out** from **synckey**.

The **seq** port is a linear sequence of 8 hexadecimal digits, e.g. **32'h12345678**. Each digit is 4 bits long. For now, you can connect this value directly to the **seq** port, hardcoding it, but later we'll create a register to store this value, updated by button presses, so that we can have different combinations.

The digit you should be checking against is dependent on the state, so if you're still in **LS0**, the digit to be compared against is the top 4 bits of **seq** (assuming you entered the digits from left to right, in which case a shift register's last entered digit would be the rightmost digit).

Once you reach LS7, and you press 8 (which matches the last digit of **seq**), you should enter the OPEN state. At this point, you should be able to relock the lock by pressing W, which returns you to **LS0** so that you need to re-enter the whole sequence to unlock the lock.

If at any point during LS0-LS7 you press the wrong button, you should enter the ALARM state and **stay there**, regardless of any button presses. To get out of it, you need to assert the top-level reset by pressing 3, 0, W.

The **state** output should always reflect the **current state** of the state machine, which will update depending on the last pressed button. The values of the states should be defined using the **typedef enum** syntax discussed in lecture, as follows:

```
typedef enum logic [3:0] {
 LS0=0, LS1=1, LS2=2, LS3=3, LS4=4, LS5=5, LS6=6, LS7=7,
 OPEN=8, ALARM=9, INIT=10
} state_t;
```

Put this definition **outside** the module. This allows you to use the **state_t** definition in any other modules where it may be helpful. For example, it allows you to change the **state** type to **state_t** in the **fsm** port header as shown:

```
output state_t state
```

Or you could use the defined states to compare against the **state** in other modules, e.g. when entering the sequence, the state will be **INIT**, so we want to see the sequence, but once the sequence has been entered and we are in **LS0-LS7**, we should hide the sequence and expect the user to re-enter it without seeing it.

To visually test this module, make the necessary connections in **top** (while connecting **state** to LEDs/an **ssdec** instance so that you can see the current state), press the corresponding buttons for the sequence to ensure that the FSM reaches the OPEN state, relocking the lock, and then pressing non-corresponding buttons to see it enter the ALARM state, which you should not be able to exit from unless you assert the 3-0-W reset.

2.2 Implementation

For your testbench, you have two choices:

1. You can make a testbench that functions like the physical FPGA, i.e. you "press buttons" and instantiate **synckey** to produce the necessary **strobe** and **out** to be used for the FSM module, or;
2. You can directly create, and set values to, your own **strobe** and **out** to be connected to the **fsm** module.

Doing the first may sound a little easier because it automatically handles the timing for you (you can simply wait for **posedge strobe** and then check the state machine output), but then you have to create, toggle, and keep track of, a separate **clk** signal for **synckey** that you can't use in the **fsm** module, in order to make **strobe** work.

Doing the second gives you more control of the signals, but you should take care to keep the operation consistent, i.e. your **strobe** should go high only after a push button has been pressed, and low only after all pushbuttons are released. Our suggestion is to use a **task** to do this (**press_button**, or similarly named, and yes, tasks can take arguments).

Regardless of which method you choose, you should be testing the following:

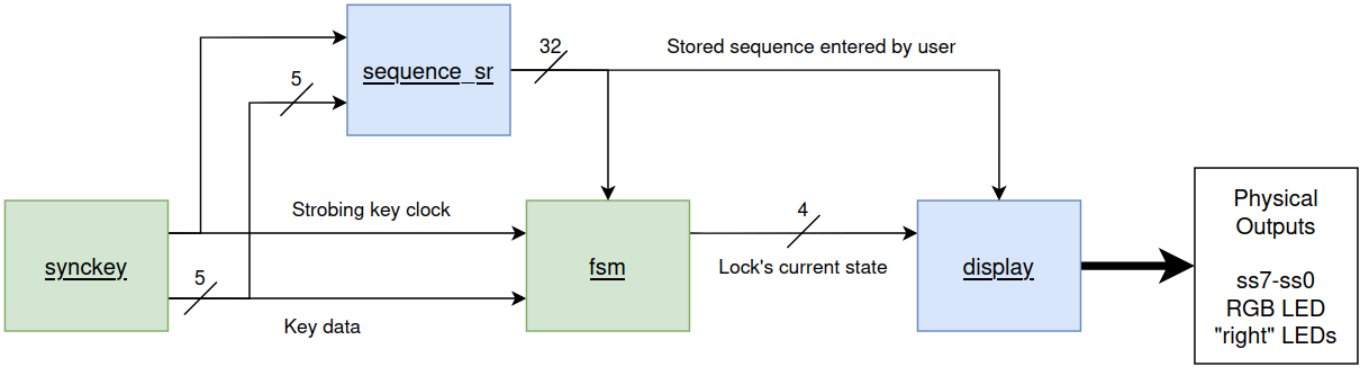
- The power-on reset case where your state machine should start from INIT.
- The regular operation, where:
  - Pressing W/**pb[16]** should cause your **fsm** to go from **INIT** to **LS0**. (The user has finished entering their sequence and would like to save/confirm it.)
  - Pressing the push buttons 1, 2, 3, 4, 5, 6, 7 and 8 would move the state from **LS0** to **LS7**. Once **LS7** is reached and 8 is pressed, the new state should be **OPEN**.
  - Relock the lock with W, which causes the state to go from **OPEN** to **LS0**.
  - Test a random combination, preferably one that makes a mistake in the middle of the correct sequence, like 1,2,3,4,6. The 6 is pressed when the FSM is in **LS4**, which is incorrect (it should be 5), and so it will go to the **ALARM** state. Assert reset here to go back to **INIT**.
- The post-operation reset where, at some point during the combination verification, the reset is asserted. This should return the FSM from whatever state it is in, back to **INIT**.

Make sure to copy the same **typedef enum** into **tb.sv** - since it is a separate file from **top.sv**, it may not find the original definition even if it was placed outside the **fsm** module.

Checkpoint L7C2: **Demonstrate** your new **fsm** module and the testbench you implemented to verify it **to your peer mentor/instructor**. Show that the testbench works for your partner's **fsm** and vice-versa. Show how your testbench cycles through each of the states, and handles all combinations of inputs for each state (OPEN should not respond to any button other than 16, ALARM should not respond to any button other than 3-0-W reset, etc.). **Make sure to document everything in your lab journal.**

3: Implement and verify the rest of the digital lock

Let's build out the rest of the lock. Here's the full block diagram for the other modules you should implement.



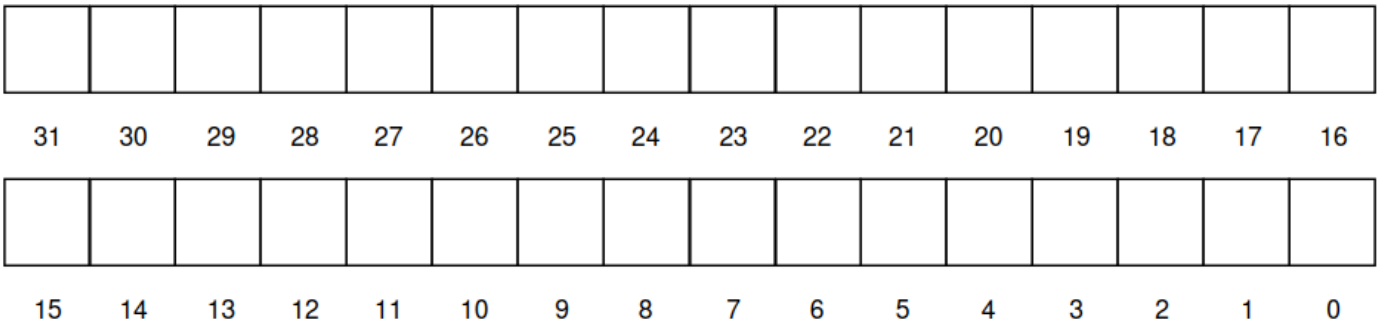
Now that you have **synckey** and **fsm**, you need to implement **sequence_sr** and **display**. **sequence_sr** will store the initial sequence that will be compared against in **fsm**, and **display** will show various things depending on the state.

Pick one of you to work on the designs, and the other to work on the testbenches. You can also mix-and-match, i.e. one of you does **sequence_sr** and the testbench for **display**, and vice-versa.

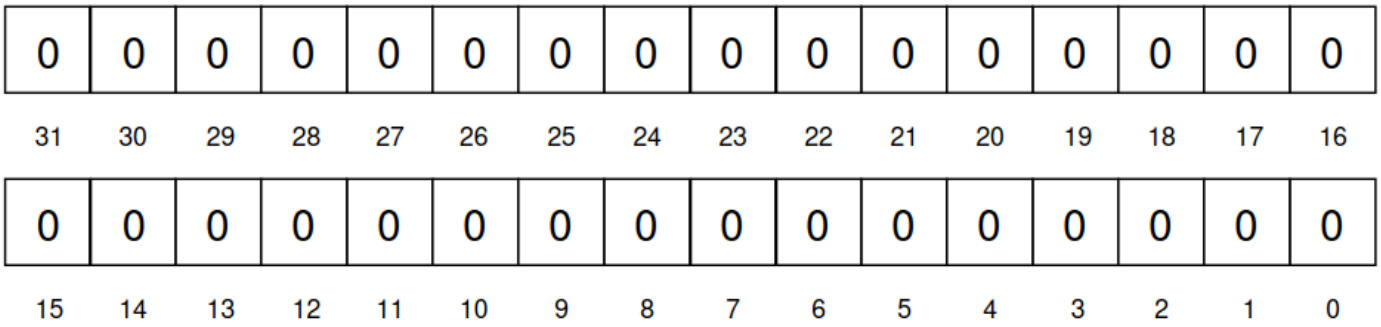
3.1. sequence_sr

Port	Type	Length
clk	input	1
rst	input	1
en	input	1
in	input	5
out	output	32

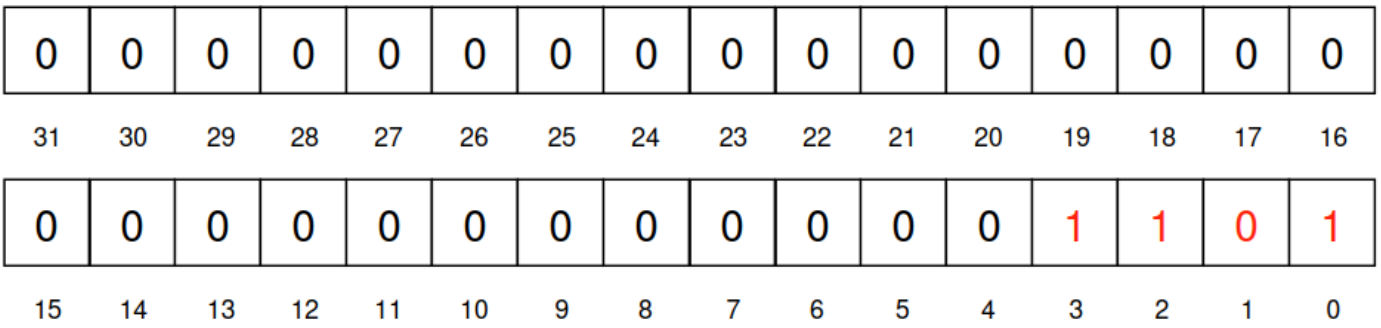
To better understand the purpose of **sequence_sr**, here is a visual representation of the output **out**, which should be connected to a 32-bit shift register. The boxes represent the value of each bit of **out**, with the number indicating the bit index.



When reset, the shift register takes on the value 32'd0, which means all bits of the shift register will be 0.

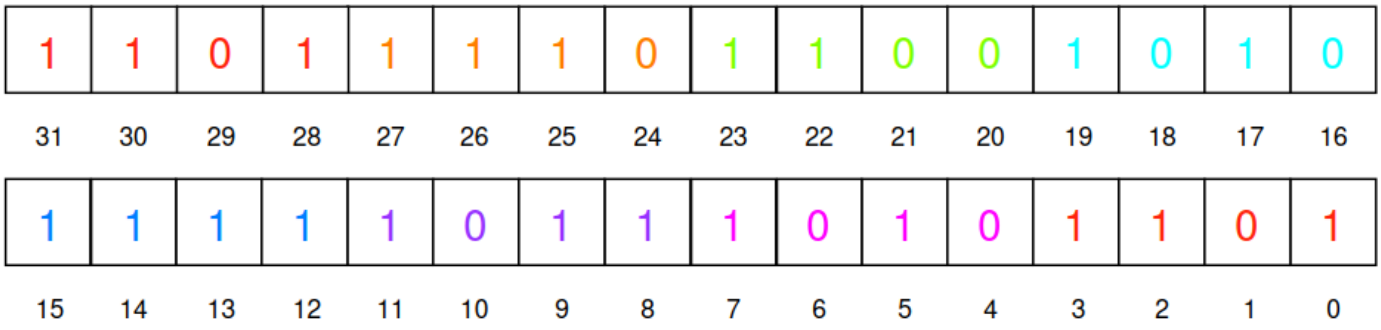


On the next rising edge of the clock, if **en** is high, the shift register will shift left by 4 bits, so that the data in 31-28 is lost, and the new data from the lower 4 bits of **in** is shifted into the register. So, if **in** was 13 (4'b1101), the shift register will look like this:



Therefore, the value of the shift register output is now 32'd13, or 32'hD. This should be the effect of pressing "D" on the physical FPGA.

Following D, if we press E, C, A, F, B, A and D, our shift register will look like this (different colors are used to indicate the value of each pressed button):



Therefore our combination is now an 8-digit sequence 32'hDECAFBAD!

If you press a ninth button, you should expect to lose the first button you pressed - this is normal.

The enable for the module should be carefully constructed. You should only change the register when in the **INIT** state. Once the lock enters **LS0**, we need to maintain the same sequence for comparison and not accidentally change it when buttons are being entered. We also should not shift in digits W-Z (16-19).

Your **sequence_sr** should not add buttons W-Z to the 32-bit register. W is submit/relock, and we're not using the other buttons. We limit ourselves to 0-F so that we can use **ssdec** to display these entered digits. Note that **in** is 5 bits long so that you can tell the difference between W (5'b10000) and 0 (5'b00000).

Visually test your **sequence_sr** on the FPGA by instantiating it in **top** along with **synckey**'s **strobe** and **out** to the respective inputs on the shift register, entering digits and viewing them with 8 **ssdec** instances connected to every 4 bits of **out**. Pressing W-Z should not affect the register.

3.2 display

Port	Type	Length
state	input	4
seq	input	32
ss	output	64
red	output	1
green	output	1
blue	output	1

This module is fairly straightforward since it is purely combinational, but functions as the **output logic** to the lock's state machine.

The **ss** bus is the concatenation of all the seven-segment displays, so in the **top** module instantiation, it will be connected to {ss7, ss6, ss5, ss4, ss3, ss2, ss1, ss0}, thereby including the decimal points, which we will assign values separately from **ssdec**, which, if you remember, only handles the **seven segments** of each display (**ssX[6:0]**).

Depending on the state, we want to display the following:

- When in the **INIT** state, we should see the output of the **sequence_sr** register, **seq**, on the 8 seven-segment displays.
- When in any of the **LS0-LS7** states, the seven segment displays should be off, and you should use the decimal points to indicate which digit should be entered. So if we are in **LS2**, the decimal point of **ss5** should be lit (**ss5[7]**). This can be used as a "hint" to tell the user what button needs to be pressed next. The blue LED should be turned on to indicate that the lock is currently **secured**.
- When in the **OPEN** state, you should display the word "**OPEN**" on the seven segment displays (right-aligned so that **ss0** displays "n"), and the **green** LED in the RGB should be lit. (**ssdec** can't do this, so you'll have to hardcode the seven-segment pattern for "OPEN").
- When in the **ALARM** state, you should display the word "**CALL 911**" on the seven segment displays, and the **red** LED in the RGB should be lit. (**ssdec** can't do this, so you'll have to hardcode the seven-segment pattern for "CALL 911").

To see our version of the lock, run **make demo**, which will flash a black-boxed version of our lock on to the physical FPGA. To try this on the simulator, you can copy in the contents of **blackbox_lock.v** included with the template into the file, and instantiate the lock inside **top** with the following:

```
lock l (.hz100(hz100), .reset(reset), .pb(pb), .left(left),
.right(right), .ss7(ss7), .ss6(ss6), .ss5(ss5), .ss4(ss4),
.ss3(ss3), .ss2(ss2), .ss1(ss1), .ss0(ss0), .red(red),
.green(green), .blue(blue));
```

Checkpoint L7C3: **Demonstrate** your new **digital lock to your peer mentor/instructor**, along with the **sequence_sr** and **display** modules and their testbenches. Show that the testbench works for your partner's modules and vice-versa. **Make sure to document everything in your lab journal.**