Lab 1 – Interfacing with Serial Keyboard

In this lab you are going to design a circuit and interface it with an external serial keyboard. Again, we are only using the simulator for this lab. Unlike in the first Verilog exercise that you can find on the student portal, we only provide the high-level problem description, and you will design the system from the scratch. We will also provide one possible design partitioning suggestion; however, feel free to come up with your own design. You can design at any level of abstraction; you are particularly encouraged to use behavioural modeling. Finally, you are required to verify your design using the provided test bench.

Requirements:

- The know-how gained when doing the first exercise, and the lectures on Verilog.

Tools:

- Xilinx Vivado Design Suite

Intended Learning outcomes:

- Mastering problem analysis and design partitioning
- Real-world RTL design example (interfacing with an external device)

Assessment:

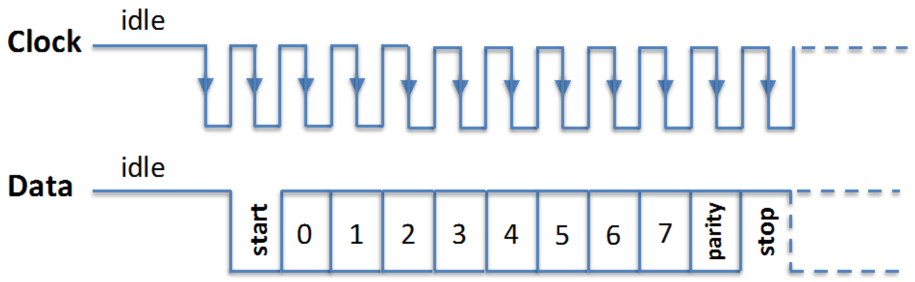
- Verify your design using the provided test bench.
- You have to demonstrate your solution, and explain your design to the lab assistants in one of the following lab sessions:
    - Thursday, April 25th, 13:15 – 17:00, room 2315
    - Thursday May 9th, 13:15 – 17:00, room 2315
- Submit the Verilog code for your solution afterwards on the student portal.

Simplified Serial Keyboard Protocol

You can read details about the serial keyboard communication protocol in:

http://www.burtonsys.com/ps2_chapweske.htm
http://uhaweb.hartford.edu/froehlich/courses/ee333/PS2key/Kb_edited.html

For the purpose of this lab, however, we assume a simplified version of the protocol. We assume a uni-directional communication from keyboard to the host. We also assume that only a single character is sent from the keyboard when a key is released, i.e. holding down a key does not send any characters (in the real protocol, separate characters are sent per key press and release, i.e. scan and break codes.)

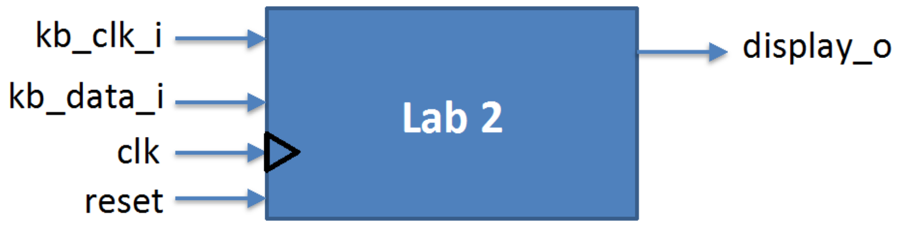


Communication is performed using two signals, clock and data. When keyboard is idle, both the clock and the data signals have the logic value "high" (logic 1.) Before sending a character, keyboard first starts generating the clock signal. Keyboard then initiates the communication by sending the "start" bit. This is done by keeping the data value "low" (logic 0) for one clock cycle.

N.B. Keyboard always changes the data values on the rising edge of the clock. Therefore, the host should read the valid keyboard data on the falling edge of the keyboard clock. Each data value remains valid for one clock cycle.

Keyboard then sends the ASCII code of the character (8 bits) starting with least significant bit, followed by a parity bit used for error detection/correction. Finally, the keyboard sends a stop bit (logic 1) that signals the end of the current transmission. We are not using the parity bit for this lab; therefore parity bit and stop bit are discarded after they are received.

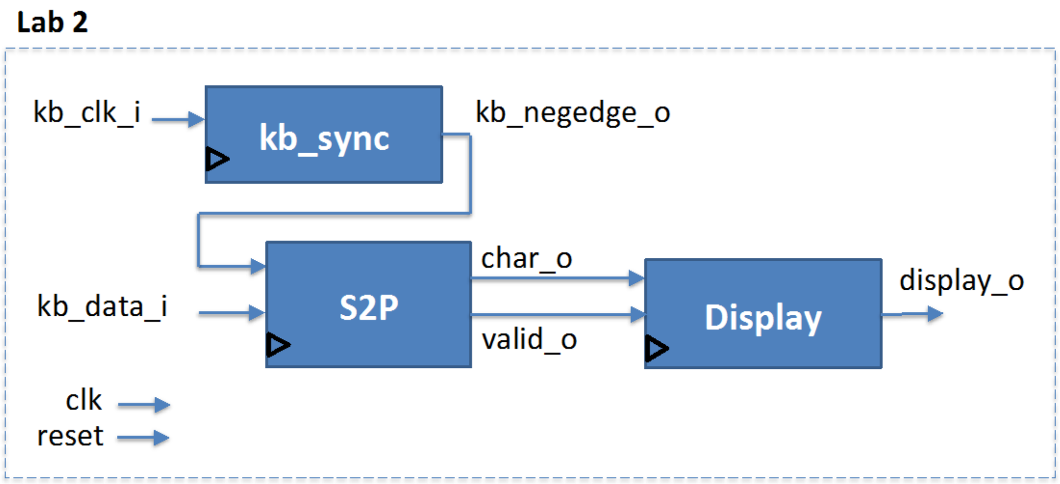
Your task is to design a module that receives characters from a keyboard and displays them. Your module should contain the following ports:



Note the difference between the “kb_clk_i” and “clk” signals. kb_clk_i is the keyboard clock used for synchronizing the data transfer from the keyboard, whereas the clk is the host system clock that is used as the reference clock in your module to implement the sequential part of your logic.

N.B. only use the “clk” signal as the clock for all your sequential parts. Connect the “clk” and “reset” to all the sequential parts of the design and make sure that all the sequential parts are reset to “0” when “reset” is asserted on the rising edge of the “clk.”

You are encouraged to come up with your own design. One possible example would be as follows:



kb_sync

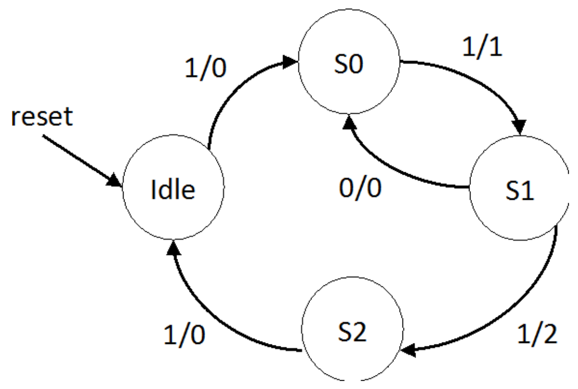
Detects the falling edge of the “kb_clk” signal. This can be done by sampling the “kb_clk” signal on every rising edge of the system clock “clk” and compare the value with the previous sample. “kb_negedge_o” is asserted whenever “preveious_value == 1’b1” and “current_value == 1’b0.”

S2P

Communication between the keyboard and the host is a serial communication. The Serial-to-Parallel (S2P) serially receives the bits from the keyboard. S2P module receives the “start” bit that marks the beginning of the data transmission, followed by 8 data bits and a parity”. Finally, the “stop” bit marks the end of the transmission. S2P discards the “parity” and “stop” bits and extracts the ASCII code of the pressed key and sends the character (b bits) to the display module. When the character is ready, the “valid_o” signal should be asserted for one clock cycle, which signals the “Display” module to read the valid character.

S2P can be implemented using a shift-register and a state machine. Shift-register is used to receive the bits serially (from LSB to MSB) and to extract the 8 bits that correspond to the ASCII code. The sequence of reading the bits serially can be controlled using a state machine.

State machines can be used to solve a variety of problems. A state machine is a memory that holds the current state of the system and transitions into new states according to the events (changes in input values.) You can then decide which operations should be performed in each state (i.e. how output signals should change in each state.) State machines can be used to solve a variety of problems, e.g. to generate the control signals needed to move data along a data path (for instance to generate the control signals in each pipeline stage.) You can read about state machines both in the course book and online sources. Nevertheless, we provide a simple example:



Assume the four states shown in the figure. Events (i.e. input values) are evaluated on the rising clock edges. Based on the current state and the event, the system either transitions into a new state on the rising clock edge, or remains in the current state.

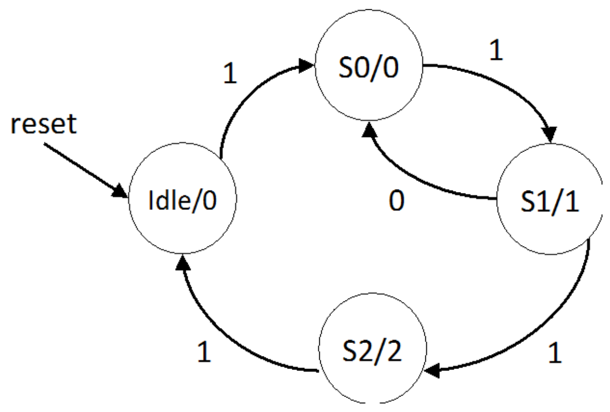
Besides state changes, the output values may also change upon changes in the input values. For instance, input value "1" in state "idle" results in transition into state "S0" and the output will be set to value "0". Likewise, input value "1" in state "S1" results in transition into state "S2" and the output will be set to value "2."

A state machine can be designed using two different approaches depending on how outputs react to changes in the inputs:

- **Mealy Machine**: outputs react immediately to the changes in the inputs without waiting for the next rising clock edge, i.e. outputs depend on the current state and the input values.
- **Moore Machine**: outputs do not immediately react to the changes in the inputs; outputs wait for the rising clock edge and change upon state transitions, i.e. outputs only depend on the current state.

The state machine shown above represents a Mealy machine, since outputs immediately change when input values change. The first value on each arc represents the input value, and the second value represents the immediate change of the output value (input-value/output-value;) however, the states change on the rising edge of the clock.

The Moore equivalence of the above machine is as follows:



Note that Moore machine may require more states. However, Moore machine removes the glitches from the output values and does not result in race condition. An example Verilog representation of the machines is as follows:

```
module Moore_Machine (clk,
                      in,
                      reset,
                      out);

input   clk, in, reset;
output  [1:0] out;
reg     [1:0] out;
reg     [1:0] state;

parameter
  S0 = 0, S1 = 1, S2 = 2, idle = 3;

always @ (state)
begin
  case (state)
    S1:
      out = 2'b01;
    S2:
      out = 2'b10;
    default:
      out = 2'b00;
  endcase
end

always @ (posedge clk)
begin
  if (reset)
    state <= idle;
  else
    case (state)
      idle:
        if (in == 1’b1)
          state <= S0;
      S0:
        if (in == 1’b1)
          state <= S1;
      S1:
        if (in == 1’b0)
          state <= S0;
        else
          state <= S2;
      S2:
        if (in == 1’b1)
          state <= idle;
    endcase
  end
endmodule
```

```
module Mealy_Machine (clk,
                      in,
                      reset,
                      out);

input   clk, in, reset;
output  [1:0] out;
reg     [1:0] out;
reg     [1:0] state;

parameter
  S0 = 0, S1 = 1, S2 = 2, idle = 3;

always @ (state or in)
begin
  case (state)
    S0:
      if (in == 1’b1)
        out = 2'b01;
      else
        out = 2’b00;
    S1:
      if (in == 1’b1)
        out = 2'b10;
      else
        out = 2’b00;
    default:
      out = 2'b00;
  endcase
end

always @ (posedge clk)
begin
  if (reset)
    state <= idle;
  else
    case (state)
      idle:
        if (in == 1’b1)
          state <= S0;
      S0:
        if (in == 1’b1)
          state <= S1;
      S1:
        if (in == 1’b0)
          state <= S1;
        else
          state <= S2;
      S2:
        if (in == 1’b1)
          state <= idle;
    endcase
  end
endmodule
```

Note that state machines can be modeled using different coding styles than the ones shown here. Also note that design tools may infer state machines from your design, even though you do not explicitly design a state machine.

Display

Display module reads a new character when the “valid_o” signal is asserted. In essence, display module is a register, which could also be integrated inside the S2P module.

Your Task

Complete the design and validate it with the provided test bench. Add the test bench files to your project by choosing “Add Sources → Add or create simulation sources.” The test bench provides the clock and reset generators, as well as the keyboard emulator. The keyboard emulator sends out “Accelerator – 1DT109.” Instantiate your design in the test bench (DUT) and verify that it receives and displays the message. You may need to declare wires in order to connect the components.