## Contents

1. **HLAB2: Prelab Exercise** 3  
   1.1 Introduction ................................................. 3  
   1.2 VERILOG Design .............................................. 5  
   1.3 Exercises ..................................................... 6
1 HLAB2: Prelab Exercise

Prelab Exercise: Design assignment - 10 marks

Aim: To design a deserialiser or serial to parallel converter: a device that decodes a serial bit stream to a byte representing a character.

1.1 Introduction

In the RPC (reverse polish calculator) and the WAT (wireless audio terminal) projects a PC keyboard will be used to communicate with the FPGA. The Spartan 3E starter board comes with a PS/2 port just like that used on a PC. Using the PS/2 port, a keyboard can transmit characters to the FPGA. PS/2 is a serial protocol similar to that used by the RS232 serial ports (Universal asynchronous receiver transmitters, UARTS) on PCs. Though serial ports are all but extinct in the PC world they are still used widely in embedded systems. The importance of serial protocols is that, just like bit serial sequential circuits, they allow data to be communicated over a single wire. This makes serial a natural candidate for communications over single wire systems such as telephone lines. Since parallel wires or multi-bit buses are used in computer systems, there is a need for a serial-to-parallel converter or deserialisers that convert from serial to bytes.

Serial communications can be either synchronous or asynchronous. Unfortunately the terms applied here are confusing. In this course synchronous means synchronised to the system clock. In synchronous serial communications the data is clocked precisely by the transmitter. For reception the receiver must regenerate the clock using the properties of the received data. Synchronous serial communications is ubiquitously used in wireless communications and Ethernet because it is good for transmission at high data rates over noisy channels. The compromise is the added complexity of requiring a clock recovery circuit at the receiver.

The serial protocols used for communications between computers on slow, low noise channels are most often asynchronous. In asynchronous serial communications, the receiver and the transmitters agree on the protocol and the data clock frequency (baud rate) prior to transmission. The protocol contains information that the receiver needs to decode the data. A typical scenario is where 8 data bits are sent over the wire preceded by a start bit, possibly followed by a parity check bit and a stop bit. The receiver detects the start bit and begins sampling the data using a pregenerated local clock to time the samples. The parity check bit simply represents the XOR of all the data bits such that the XOR of the data with the parity check gives one (the PS/2 protocol has odd parity). If a single bit is incorrectly detected then the parity check produces a 0 implying an error. The

---

1Note that we are talking about single wire communications here. Many computer to computer communications protocols also use multiwire communications also referred to as serial where the data and the data clock are sent on different wires: a good example are protocols using the parallel port, serial peripheral interface (SPI) and I2C protocols.
parity check can be used to decide if action should be taken such as dropping the frame or requesting a repeat transmission. In lectures we had a brief look at cyclic redundancy checks (CRCs). CRCs are much more effective at bit error detection than single parity check bits. CRCs are used in Ethernet.

An example of an asynchronous serial protocol is RS232: applicable to PC serial ports. One of the common communications conventions in RS232 is 8N1 in which there are 8 data bits, no stop bit and 1 parity check bit. In addition there will always be an idle state in which the voltage on the line takes a default value in the absence of communications and a start bit which designates the arrival of data. Unlike synchronous serial communications where very large bit trains can be transmitted without interruption, the difficulty of maintaining a coherent clock for long periods in asynchronous comms means that data is transmitted as single frames (often 8 bits) at a time. This is highly error prone because the entire frame is lost if the start bit is misdetected.

Fig. 1 shows a trace of the PS/2 protocol bit train. In the present exercise you will design the VERILOG to decode the bit sequence using the idle, start and stop bits to delineate the data bits.

Fig. 2 shows how the number E5 (Hex) or 229 (decimal) would be transmitted in serial for the case of PS/2. Note that the lowest order bit occurs earliest in time, right after the start bit.

Figure 1: PS/2 serial protocol

Figure 2: A PS/2 serial bit stream and clock coming from a keyboard. The bit stream transmits the sequence 10011011 representing the number 9B (Hex) or 155 (decimal). Note the reverse order between how the binary sequence is written and how it appears on the trace.
PS/2 Communications: According to the signal traces of Fig. 2, serial transmission by the keyboard using the PS/2 protocol proceeds as follows. When data is to be transmitted the IDLE is set HI and the CLOCK ticks such that the falling edge (negedge) of the clock coincides with the centre of a bit. *Bits are only valid at the falling edge of the clock.* To initiate transmission of DATA bits, a START bit is transmitted. The START bit is a LO that follows the HI of the IDLE state. In your code you have to detect this transmission from the IDLE state to the START bit. The DATA is then transmitted bit by bit with LSB (least significant bit) transmitted first. The PARITY bit is transmitted at the end of the DATA bit stream. Its value is given by $\sim(\bar{\text{data}})$. The PARITY bit is followed by the STOP bit which is a HI. Detection of a HI at the position of the STOP bit is the defining condition of a successful transmission. If a LO STOP bit is detected then the frame is dropped as garbage. Note that the return to the IDLE state now is unnecessary until the next transmission occurs however by the arrival of the bit after the STOP bit the data CLOCK has probably been halted.

The following website explains in detail how a keyboard works.

http://www.beyondlogic.org/keyboard/keybrd.htm

1.2 VERILOG Design

Fig. 3 shows the block schematic of a deserialiser.

![Block schematic of a deserialiser](image)

**Figure 3:** Block schematic of a deserialiser

**Inputs:** There are two inputs, the data and the data clock. Even though PS/2 is an asynchronous protocol we have the luxury of a clock provided by the keyboard. - PS/2 is a multi-wire protocol providing the PS/2 data clock - *unlike RS232 communications you do not need to generate the clock in your VERILOG.* The outputs represents the one byte transmission from the Keyboard. The keyboard may not generally change its data line on the rising edge of the clock as shown in the diagram. The data line only has to be valid on the falling edge of the clock. The keyboard generates the clock. The frequency of the clock signal typically ranges from 20 to 30 Khz. The Least Significant Bit is always sent first.
Data is sampled on the **negedge** of this clock. An important detail is that all of our designs have to be synchronous to the FPGA clock. This means that in practice the actual clock used to sample the data will not be that of the keyboard but a new clock generated by the FPGA (your VERILOG) from both the keyboard clock and the FPGA system clock. This can be achieved by code similar to that shown below.

```verilog
module ps2_clk_sync(sys_clk, ps2_clk, ps2_clk_syn);
  input sys_clk;
  input ps2_clk;
  output ps2_clk_syn;
  reg ps2_clk_syn;
  always @(posedge sys_clk) begin
    ps2_clk_syn <= ps2_clk;
  end
endmodule
```

In the prelab exercise you may generate your deserialiser input clock, `clk`, by a test bench. However we will need to employ the above clock sync code in the hardware lab.

**Outputs:** The deserialiser returns the data byte (8 bits), `data`, a 1-bit flag, `done` indicating that detection has completed so that the connecting hardware can read the `data` and the 1-bit parity check `pc`. Letting the data be `[7 : 0]data` and the parity bit be `P_b` then $pc = (\hat{data} \hat{P_b})$.

**Architecture:** The deserialiser is a shift register just like the CRC polynomial divider. Fig. 4 shows how this works. The shift register is therefore just a bunch of concatenated D flip-flops for which you already know how to code the VERILOG.

![Figure 4: A shift register](image)

### 1.3 Exercises

**Exercise 1 (7 marks):** Complete the code by using the idle state, start and stop bits according to Fig. 2 to determine the `data`, assign `done` and compute the parity check `pc`.

**Hint:** Let your module i/o line look as follows,

```verilog
module serial_rcvr(clk, in, data, done, pc);
  input clk;
```
Define an input bit stream \textit{in} that is synchronous with the \textit{clk} and an output byte variable \textit{reg [7 : 0] data} to store the final result and a temporary storage variable \textit{reg [10 : 0] tmp} large enough to contain a complete frame. Your VERILOG should contain an \texttt{always @}\texttt{(posedge clk)} block as usual for describing D flip-flops that contains a set of statements assigning the elements of \textit{in} in sequence to \textit{tmp}. There will also be a conditional statement to detect \texttt{START} bits and then assign \textit{data}, \textit{done} and compute the parity check \textit{pc}). The purpose of the temporary storage \textit{tmp} is so that the entire bit train can be checked in one time step for the essential characteristics of the serial transmission such as \texttt{IDLE} to \texttt{START} transition etc.

```
always @(negedge clk) begin
    tmp[10] <= in;
    tmp[9] <= tmp[10];
    ...
    ...
    tmp[1] <= tmp[2];
    tmp[0] <= tmp[1];
    if(....) begin
        //Assign done, data, compute pc,... etc.
    end else begin
    end
end
```

\textbf{Exercise 2 (3 marks)}: Simulate your code using the following test bench. Produce the GTKwave traces and paste them in your logbook with explanation.

\textbf{Hints}: First study the test bench to make sure you understand how it produces the input to your deserialiser `\texttt{serial}_{rcvr}.v'`. Then look at the traces of Fig. 5 and try to understand how it works. Note that the data train being sent is exactly that in Fig. 2.

```
module TB_serial_rcvr;
    reg clk;
    reg in;
    wire [7:0] data;
    wire done;
    wire pc;
    reg [8:0] inputs;
    serial_rcvr sr(clk, in, data, done, pc);
```

initial begin

////////////////////////////////
//Idle state
clk = 1'b0;
inputs = 9'b010011011; //9th bit is parity
#1in = 1'b1;

////////////////////////////////
//Start bit
#1clk<=1'b1;
#1clk<=1'b0;
#1$display("A%d	%b	%b	%b	%d",$time,in,data,done, pc);
////////////////////////////////
//Data
#1clk<=1'b1;
#1clk<=1'b0;
#1$display("B%d	%b	%b	%b	%d",$time,in,data,done, pc);
////////////////////////////////
//Bit 0
#1in <= 1'b0;
#1display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);
#1clk<=1'b1;
#1clk<=1'b0;
#1$display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);
#1clk<=1'b1;
#1clk<=1'b0;
//Bit 1
#1in <= inputs[1];
#1$display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);
#1clk<=1'b1;
#1clk<=1'b0;
//Bit 2
#1in <= inputs[2];
#1$display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);
#1clk<=1'b1;
#1clk<=1'b0;
//Bit 3
#1in <= inputs[3];
#1$display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);
#1clk<=1'b1;
#1clk<=1'b0;
//Bit 4
#1in <= inputs[4];
#1$display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);
#1clk<=1'b1;
#1clk<=1'b0;
//Bit 5
#1in <= inputs[5];
#1$display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);
#1clk<=1'b1;
#1clk<=1'b0;
//Bit 6
#1in <= inputs[6];
#1$display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);
#1clk<=1'b1;
#1clk<=1'b0;
//Bit 7
#1in <= inputs[7];
#1$display("0%d	%b	%b	%b	%d",$time,in,data,done, pc);

8
#1clk<=1'b1;
#1clk<=1'b0;

// Parity
#1in <= inputs[8];
#1$display("P%d	%b	%b	%b	%d", $time, in, data, done, pc);

// ______________________________
#1clk<=1'b1;
#1clk<=1'b0;

// Stop
#1in <= 1'b1;
#1$display("8%d	%b	%b	%b	%d", $time, in, data, done, pc);

// ______________________________
#1clk<=1'b1;
#1clk<=1'b0;

// ______________________________
#1clk<=1'b1;
#1clk<=1'b0;

// ______________________________
#1clk<=1'b1;
#1clk<=1'b0;

// ______________________________
#1clk<=1'b1;
#1clk<=1'b0;

end
initial begin
$dumpfile("TB_serial_rcvr.vcd");
$dumpvars;
end
initial begin: stopat
#200; $finish;
end
endmodule

Figure 5: GTKWAVE output from the above simulation