

**University of California at Berkeley**  
**College of Engineering**  
**Department of Electrical Engineering and Computer Sciences**

EECS 150

Spring 2002

**Lab 7 (Project Checkpoint #1)**  
**UART Design**

## **1 Objective**

In this lab you will design, simulate, implement, and test the receiver portion of a *universal asynchronous receiver / transmitter* circuit (UART). This circuit will be useful later for your course project, but more importantly, through its design you will gain more experience with the FPGA device.

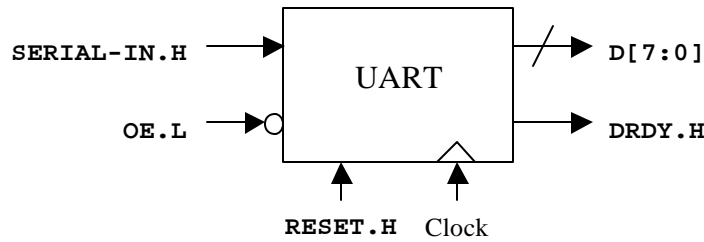
## **2 Functional Specification**

UARTs are used for communication between two devices. For instance, UARTs are used to connect terminals to computers. UARTs are also used in systems using MIDI. They provide a means to send data with a minimum of wires. The data is sent bit-serially, and no clock signal is sent along with it. The primary function of a UART is parallel-to-serial conversion when transmitting, and serial-to-parallel conversion when receiving. The fact that a clock is not transmitted with the data complicates the design of a UART. The two systems (sender and receiver) have separate, unsynchronized, clock signals. Although the two clocks will have the same frequency, they will not have the same phase. Part of a UART's function, and the tricky part, is to "sample" the serial input at just the right time to reliably capture the bit stream. A high-speed clock to sample the bit stream multiple times per data bit allows one to accomplish this task.

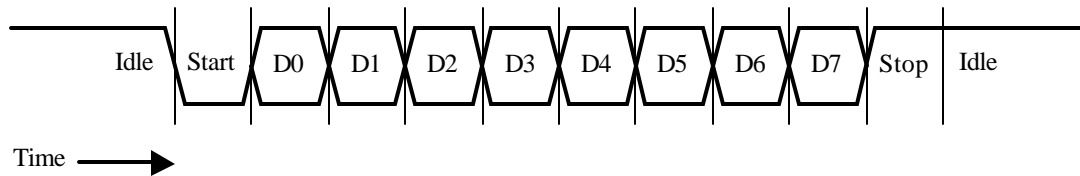
In our application the bit transfer rate, or *baud rate*, is 31.25kHz. This rate is the standard for MIDI devices and will be useful later for the course project. Your FPGA board is equipped with a 16MHz crystal oscillator. To generate a 31.25kHz clock signal divide the crystal oscillator frequency by 512. We do not actually need that frequency for this lab. The frequency we need is 8 times the MIDI baud rate:

$$8 \times (31.25\text{kHz}) = 250\text{kHz}.$$

Figure 1 shows a functional block diagram of the UART receiver that you will design. Bit-serial data is received on the `SERIAL-IN.H` input. When one byte of data has been received, it is output to the `D` output bus, and the output control signal `DRDY.H` is asserted for one clock period. The block is clocked with a frequency 8 times the baud rate, in this case 250kHz. The reset signal `RESET.H` and output enable `OE.H` are optional.



**Figure 1:** UART symbol.



**Figure 2:** A MIDI transmission character.

Data is transferred one byte at a time to the receiver using the format shown in Figure 2. The transmission character is composed of an 8-bit data byte, sent LSB first, preceded by a start bit (LOW) and followed by a stop bit (HIGH). When no character is being transmitted, the line is idle (HIGH). The line need not go idle between characters, as it is possible for the start bit of a transmission to immediately follow the stop bit of the previous transmission.

### 3 Theory of Operation

This section describes the internal operation of the UART receiver that you will design. There are many possible detailed designs – we will not present them all here, but we will give you some ideas.

The simplest solution for receiving transmission characters is to generate a clock signal at the baud rate, in this case 31.25KHz, and to use it to clock the bits one at a time into a serial-to-parallel converter. A serial-to-parallel converter is simply a shift-register with its internal flip flops connected directly to outputs. For help on describing a shift register in Verilog, use “Language Assistant” feature in HDL Editor. In general, the following code generates a shift register that operates on the positive clock edge:

```
reg [3:0] out;
always @(posedge clk)
begin
    out = {in, out[3:1]};
end
```

The key to understanding this code is the concatenation operator, specified by curly braces. All signals enumerated inside the braces are joined together to form one “bus”.

A counter can be used to build a simple controller which counts-off the ten bits of a transmission character (start, 8 data bits, and stop), generates the data ready signal `DRDY.H`, then resets itself. Counters are available in the Xilinx FPGA component library, but *none are primitive* to XC4000X part and therefore cannot be instantiated directly. We suggest that you develop your own counter, which may include some of the features of the library counters. For example, some counters are *loadable* with an initial starting value. Non-loadable counters count from 0. Normal operation of a non-loadable counter, after being reset, is to count from 0 to  $2^n - 1$  (1 increment per cycle), then to *wrap around* and start counting at 0 again. Loadable counters are used to count through fewer than  $2^n$  values, by starting at an initial count other than zero and reaching the maximum count sooner. Some counters in the library have a *terminal count* (TC) output signal, which is set high on the cycle when the counter reaches its maximum value  $2^n - 1$ .

In this application, the actual Q values of the counter need not be used, only the TC signal. With the proper input to a loadable counter, the TC signal can be made to assert once every 10 clock cycles – exactly what we need for the `DRDY.H` signal. TC can also be used to reset the counter by enabling the load of the initial count value into the counter.

Take a look at the data-sheet for the “Serial-in Parallel-out” shift register **SR8RE** in the Xilinx Library Guide (available from Xilinx’s web site as well as our course web page). It is instructive to start by sketching out a simple UART design based on this component and a 4-bit counter. If you have the time, you might want to implement your design in Verilog and test it out in the lab. You will find is that it has unreliable behavior. Because the receiver clock is out of phase with the sender clock, it is possible for the circuit to capture input bits when they are not at valid logic values, *i.e.* during transitions.

The way we will build a reliable receiver is to start with a high-speed clock signal, in this case 8x the baud-rate clock, 250kHz. This clock is used to super-sample the input waveform. A stage is used to determine the clock period when the start bit is approximately “half-way” transmitted. This can be implemented by 4-bit serial-in parallel-out shift register with its outputs NOR-ed. The circuit generates a high output when it sees the input signal low for four consecutive clock cycles. When this happens, we assume that the input bit is the start bit, by now half-way received (recall that we have 8 250kHz clock periods per bit transmission time). This NOR-ed signal can be used to reset a counter to count off 8 clock cycles. The counter delays until approximately the middle of the first data bit, at which time we sample the input and record it by shifting into a data shift register. This will capture the first data bit in the shift register. We wait another 8 clock cycles, and then enable the shift register to capture the second data bit. These steps are repeated for 8 consecutive data bits (64 cycles). You can use a second counter to control the “outer-loop” which counts through the 8 data bits.

While many variations on this basic scheme exist, you should be able to design the receiver with two counters, two shifters, and a simple FSM controller. Remember that in a good design, all the clock inputs to the components come from the same source. Hint: If you wish to slow down the operation of a counter or shifter, use its “clock enable” input rather than trying to slow down its input clock.

## 4 Pre-lab

Before coming to lab, design an 8-bit UART receiver. You need not enter the entire design into the Xilinx HDL editor for pre-lab. You must, however, sketch a paper design from components that you plan to use (e.g. counters, shift registers) and design an FSM to control your UART receiver.

Design a clock divider circuit to produce a 250kHz signal from the Xilinx 16MHz clock and write it in Verilog. Use “Language Assistant” to help you design synchronous counters required to build the divider.

To help test your UART in lab, design a circuit to display values received by the UART on the Xilinx board’s 7-segment display (a pair of LEDs for numeric characters). The circuit uses a data register to record whatever new data is emitted by the UART receiver and outputs it to the display. The top level Verilog module, provided to you, instantiates two `bin2LED( )` components that convert a 4 bit binary number into 7-segment LED input.

For the prelab, do the following:

1. Read the data sheets for the following Xilinx library components: (available in the *Xilinx Library Guide*, linked from the course web page) **CC16CE**, **CB4CLE**, **SR8RE**, **BUFG**. Understand their operation and special inputs and outputs (e.g. CE, TC, RESET, etc.). Decide which features you may want to use in your UART design and think about how to implement them in Verilog.
2. Answer pre-lab questions on the check-off sheet
3. Sketch paper designs for the UART and clock divider.
4. Write a simulation script to simulate the UART receiving a single byte in each of the following cases:
  - (a) The line is idle after reset, then a character `0x3E` is transmitted.
  - (b) The line is idle, then a character `0xA5` is transmitted.
  - (c) The line is idle, then a burst of two bytes is received.
  - (d) The line is idle, then a burst of three bytes is received.

Your timing diagram should show clock, `SERIAL-IN.H`, any other control signals, and `DRDY.H`. Specifically, all your simulations and actual runs on the

board must show a signal indicating the points in time where your UART samples its input.

## 5 Lab Assignment

### 5.1 Verilog design and Simulation

Enter your design into the Xilinx HDL editor and simulate it as described in the pre-lab. Show your TA the simulation results. Make sure that all your Verilog files contain a header with both lab partners' names.

We have included a constraint file to take care of all pin connections. The top level Verilog file contains detailed instructions on the names of the signals to use and where to instantiate your UART module.

Your design will use pin 71 for its input and pin 3 to output "sample" signal, which shows when your UART samples its input.

### 5.2 Communication Between Two Xilinx Boards

You will physically test your design by having a pair of Xilinx boards communicate over jumper wires. One board will implement a serial transmitter (using a TA-provided schematic `sender.bit`) while the other board implements a serial receiver (using your UART design).

Physically connect `PIN 71` of the sender board to the serial input of the receiver board (should be `PIN 71` in your UART design). Also connect the grounds of the two boards together. **Before powering on the boards**, have your TA check-off the connections.

Connect the Xilinx XCHECKER cable to the sender board. Power on the sender board and download into it the TA-provided sender schematic from the class website.

Reconnect the XCHECKER cable to the receiver board. Power on the receiver board and download into it your UART design.

The sender board uses two inputs. When the `SPARE` button is pushed, the eight DIP switches on `SW5` will be sent on the serial line (surrounded by the start and stop bits). Try several different data values to verify that your UART is working.

Have your TA check-off your working UART. Your TA may select arbitrary values to send down the jumper cable. Successful competition of this test indicates that your UART receiver is capable of receiving one isolated byte of information.

Now verify that it can also handle several consecutive bytes (sent one byte right after another). Physically disconnect the wire on `PIN 71` of the sender board and reconnect it to

PIN 72. The TA-provided sender uses PIN 72 to send *bursts of three consecutive bytes*. The operation is similar: you set DIP switches to an arbitrary value, and when SPARE button is pressed, the sender board sends the values DIP, DIP+1, and DIP+2 one after another.

Finally, demonstrate that your UART receiver correctly samples incoming data by using the Logic Analyzer to show that your receiver actually samples its input (approximately) in the middle of each transmitted bit. The `uart_sample` signal is connected to PIN 3.

## **7 Acknowledgments**

Original lab by J. Wawrzynek (Fall 1994). Modifications by N. Weaver and E. Caspi. Verilog related modifications by Yury Markovskiy and Jack Sampson (Spring 2002).

Name: \_\_\_\_\_ Name: \_\_\_\_\_ Lab Section: \_\_\_\_

## 8 Check-offs

### Prelab Questions

- Read the data sheets for the following Xilinx library components: CC16CE, CB4CLE, SR8RE, BUFG
- Using the library's counters, how would you design a circuit to count a given number of cycles  $N$ , then assert a 1-bit "done" output? Draw a block diagram.

TA: \_\_\_\_\_ (5%)

- Why should the 250kHz clock be distributed to the circuit using a BUFG?

TA: \_\_\_\_\_ (5%)

### Prelab Design

- UART receiver paper design TA: \_\_\_\_\_ (20%)
- Clock divider paper design TA: \_\_\_\_\_ (5%)
- UART simulation script TA: \_\_\_\_\_ (5%)

### Verilog Design and Simulation

- Working UART simulation TA: \_\_\_\_\_ (25%)
- Show the templates that you used (if any) in the "Language Assistant"

TA: \_\_\_\_\_

### Board-to-Board Communication

- Proper wire connections TA: \_\_\_\_\_
- Working serial communication
  - 1 byte TA: \_\_\_\_\_ (20%)
  - 3 byte burst TA: \_\_\_\_\_ (10%)
- Verification using logic analyzer TA: \_\_\_\_\_ (5%)

Turned in on time TA: \_\_\_\_\_ (×100%)

Turned in 1 week late TA: \_\_\_\_\_ (×50%)