Simple UART timing device using Verilog and FPGA

Michael Kim, Umar Khan, Alan Huang[1]
[1]*PHYS-234: Digital Electronics (Dr. Brody)*

I. INTRODUCTION

For this second project, we again used a medley of equipment. This includes the Basys 2 Spartan-3E FPGA Trainer Board, a computer running Windows, the Spyder scientific environment for Python, a UART connection adapter chip, a connecting cable from the UART chip to the computer, the Xilinx ISE Design Suite, and the Digilent Adept software. Furthermore, we used Verilog syntax taken from course notes.

We set out to create a timer which first records the number of deciseconds a switch was turned on, and then transmits that data using serial communication to the computer. To accomplish this, we connected the FPGA board to our UART chip, connected the UART chip to the computer with the connecting cable, used Xilinx ISE Design Suite to write Verilog code, and finally "flashed" the generated *.bit* file to the FPGA board using Digilent Adept. Then we ran a small Python script to read incoming data bytes transmitted by the UART chip.

II. PROCEDURE

After preparing the FPGA board by connecting it to the UART chip, we connected that chip to the computer. One pin on the chip represents a transmit pin and another one represents a receive pin. The receive pin was inserted into the output "A3" on the board.

III. VERILOG MODULE CODE

We coded three inputs: a on/off input **switch** to control the timer, a timer reset button **reset**, and a clock **sclk** with frequency 50 MHz. The only output **TxD** was the pin through which we transmitted a single byte representing the number of deciseconds counted.

```
module timer(
   input sclk,
   input switch,
   input reset,
   output reg TxD
);

// first create a 32-bit binary counter
reg [31:0] counter;

// which helps us make a 19200 Hz clock
reg clk;

// by toggling clk every 1302 rises
always @(posedge sclk) begin
   if (counter == 1302) begin
      counter <= 0;
      clk <= ~clk;
   end
   else begin
      counter <= counter + 1;
   end
end

// now to implement a state machine
// create a counter for our new clock
reg [31:0] recounter;

// create a flag for the final state
reg done;

// and an 8-bit timer to transmit
reg [7:0] timer;

always @(posedge clk) begin

   // reset if reset button is pushed
   if (reset == 1) begin
      timer <= 0;
   end

   // now while switch is set to high...
   if (switch == 1) begin
      recounter <= recounter + 1;
      // increment our timer
      if (recounter == 1920) begin
         timer <= timer + 1;
         recounter <= 0;
      end
      // stay at state 0 of mealy machine
      TxD <= 1;
      state <= 0;
      done <= 0;
   end

   // and when switch is set to low...
   else if (state == 0 & done == 0) begin
      TxD <= 0; // start transmitting!
      state <= 1;
   end

   //  continue to go through states
   else if (state > 0 & state < 9) begin
      TxD <= timer[state - 1];
```

```
        state <= state + 1;
    end

    // until the final state is reached
    else if (state == 9) begin
        TxD <= 1;  // stop transmitting!
        done <= 1;
        state <= 0;
    end

end
endmodule
```

IV. VERILOG CONSTRAINTS CODE

The constraints file was programmed with the FPGA board-specific inputs and outputs. In particular, the right-most button were treated as a timer reset input, and the right-most switch was treated as a timer activator input. The UART adapter chip connected to the receive pin, which was treated as an output for Verilog.

```
net sclk loc = "B8";
net switch loc = "P11";
net reset loc = "G12";
net TxD loc = "A3";
```

V. COMMENTS ON VERILOG CODE

Clock **sclk** continuously oscillates in a step-wise manner between high and low voltage. To reach our needed baud rate of 19200 Hz, we simply create a counter **counter** which increments on each rising edge of **sclk**. If **counter** reaches a number N, then we flip **clk** and reset **counter** to 0.

$$N = \frac{(\frac{50MHz}{19200Hz})}{2}$$

It is easy to see that in our case, we flip **clk** upon **counter** reaching 1302. Then, upon rising edge of the input **switch**, we begin a separate counter called **recounter**. The idea is that we know that there should be 19200 rising edges of **clk** in one second. Then, we simply increment **recounter** upon each rising edge of **clk** until it reaches a number L. Then, we increment our 8-bit reg **timer**.

$$L = \frac{19200Hz}{10}$$

Observe that with $L = 1920$, we are effectively incrementing **timer** every decisecond (one-tenths of a second). Then, this incrementation of **timer** continues until the switch is turned off. At that point, the rest of the Verilog module is an implementation of a Mealy machine, sending us through 9 states:

```
(Begin serial transmission)
state == 0: Lower TxD to 0

(Proceed with transmission)
state == 1: TxD is timer[state - 1]
...
state == 8: TxD is timer[state - 1]

(End serial transmission)
state == 9: Raise TxD to 1
```

At the end of the transmission, we set **done** to 1 which implicitly ensures that we loop back to state 0. We also explicitly set **state** to 0 as well. Then, the timing and transmission are complete. It is a feature that **timer** is not reset to 0 after reaching state 9. As a result, it is possible to record discontinuous intervals of time. Then **timer** will only reset if the input **reset** is pressed or the value for **timer** exceeds 8 bits (255 deciseconds).

VI. PYTHON CODE

The purpose of the Python code is to continuously attempt to read data send by the UART adapter and report to the user the number of deciseconds the FPGA board switch was held down. After initializing the *Serial* object using the hardware identifier *COM8*, we ran a while-loop to read about every second. This is achieved by instructing the interpreter to sleep for one second per loop.

```
import serial
import time

com = serial.Serial('COM8', 19200)
while True:
    // read first entry in list
    print(com.read()[0])
    time.sleep(1)
```

VII. FURTHER REMARKS

We add some additional comments about the project. We chose to transmit the number of deciseconds the switch was held down instead of, say, the number of milliseconds or the number of nanoseconds. Transmitting the number of milliseconds was certainly possible due to our baud rate of 19200 Hz, which in practical terms means that we could divided a second into up to 19200 intervals. However, transmitting timing data with the fidelity of a single nanosecond is impossible due to the same limitation.

Furthermore, recording the number of milliseconds would have severely limited our ability to make sure the code was functioning properly. That is, unless we added additional code to allow the FPGA circuit to transmit

multiple bytes of data via serial communication instead of just one, the computer would at most pick up 255 milliseconds.

Although not implemented in this project, we envision a feature enabling us to transmit multiple bytes of data in one go. If successfully implemented, then transmitting millisecond data starts becoming a useful metric or we could also count a greater number of deciseconds.

One possible way to implement this would be to use a divide and conquer method. For example, if we have a n-byte timer then we can divide this into n bytes and transmit them one at a time. Then, we might reserve the first two most-significant bits of each byte to encode which byte it is among the n many bytes to send. Then, using an identifier for the first byte to transmit, we could inform the computer (which is still continuously receiving) that it is about to receive n bytes of data. Having the Python script delay receiving for a shorter interval will then allow it to receive those n bytes, then reconstruct that into a meaningful 16-bit number which relays the total number of milliseconds, nanoseconds, etc. to finally display to the user.

VIII. CONCLUSIONS

Our observations from this implementation show one of the possible outcomes of combining a Mealy machine with serial communication. The immediate application of our implementation is that we have created a circuit that can relay timing data to a Python program via UART.

In a real world application, the Python back-end can be modified to process the timing data received from the FPGA into useful graphs and diagrams. In a situation where accurate timing is less important that relative time, our circuit could be used as well. For example, a user could interface with the FPGA by transmitting a message to a computer using Morse code (which relies on the duration of switch activation), then have the Python program transmit the message to another computer.