Lab 3

Where No Clock Has Gone Before

Introduction

In this experiment, we will introduce the concept of a Finite State Machine (FSM) and use it to create a common sequential design, a Universal Asynchronous Receiver Transmitter (UART). By using an FSM to control the behavior of a circuit in a certain sequence, we will be able to produce output that can be read by a computer and displayed in a terminal emulator.

Prelab: U-what?

Background

In order to send data between two machines, wires need to be used (unless we're using fancy wireless methods). In the old days, this was done using parallel transmission where several bits were sent simultaneously (think the old IDE "ribbon" cables used in PCs). As systems grew in speed, parallel transmission started to become less and less appetizing; it turns out that having a lot of wires next to each other operating at high frequencies suffers from large amounts of cross-talk due to capacitive coupling of the wires (the wires are plates and the insulation between is a dielectric). The solution to this problem has been serial transmission. Instead of sending multiple bits at once, one bit is sent at a time. This allows us to avoid the aforementioned cross-talk and operate at higher transmission speeds.

The question then becomes how to send data over as few wires as possible. In a UART, that answer is 2 wires: one for receiving data and one for sending data. But wait, you ask, where is the clock? How do the two sides know when data is coming and how to separate that data into distinct bits? The answer is that both sides have to be set up in advance to operate at the same clock frequency (baud rate). So long as the frequency is the same, within a margin of error, the phase does not matter. In this way we say it is Asynchronous because both ends are operating on different clock domains.

So now you know WHY we use serial communication and why it's Asynchronous. The question becomes

HOW you use serial communication. In order to understand that, you have to look at the protocol.

Common serial communication uses the protocol shown below in figure (3.1):

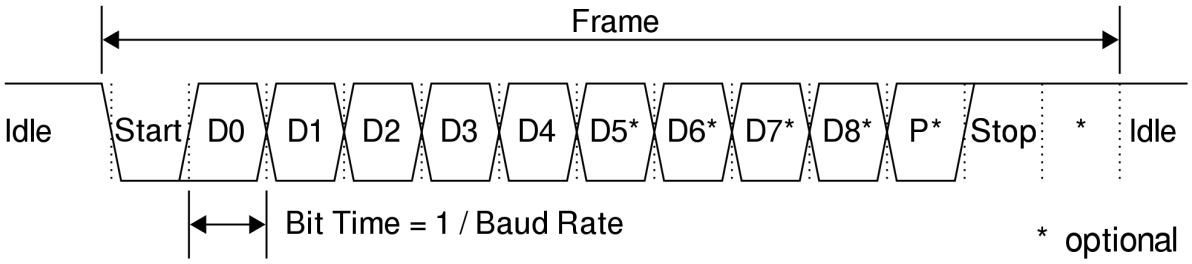


Figure 3.1: UART Frame Waveform [2]

Here is a general verbal description of the image above:

- While the line is held high, no data is being transferred and the line is in an idle state
- As soon as we sample the line and find it to be low, that indicates the start of the data transmission
- For the next [Data Bits] clock cycles, we sample the line and store the value into a shift register (LSB is transmitted first)
- After [Data Bits] cycles, an optional parity bit is sent (usually XOR of the data bits)
- Finally, [Stop Bits] bits of logic high are sent to indicate the end of the transmission

Note: This is a description of what is seen by the receiving side. To make the sending side, YOU will need to create these changes on the tx line with your design

Helping Hand

In order to guide the design of the lab, we are going to give you the design for the receive side of a UART. Use it as a reference for designing the sending part. (See end of manual for design)

Shown below is the FSM state diagram for the RX (receiving) side of the UART:

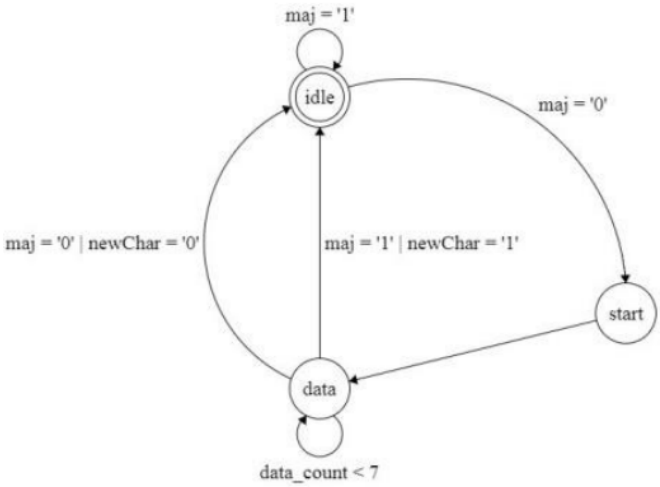


Figure 3.2: UART RX State Diagram

Prelab Task

Draw the FSM state diagram for a device that takes as input an 8-bit data packet and transmits it using the protocol in figure (3.1) with 8 data bits, 0 parity bits, and 1 stop bit. It should take as input both a clock and a clock enable signal for timing, with input control signal `send`, and output a signal called `tx` that is the serial output as well as a signal called `ready` that indicates the machine is in an idle state.

1 We can rebuild him

1.1 Tasks

1. Create an entity called `uart_tx` with the following interface that sends data using the protocol in figure (3.1) with 8 data bits, 0 parity bits, and 1 stop bit. See listing (3.1) for entity description. It should have the following behavior:
   - When `rst` is asserted, all internal registers are cleared and it goes into an idle state
   - When it is in the idle state, `ready` is **1**
   - When `send` is asserted and enable is **1** and the clock is on a rising edge, it stores `char` into a register and begins the sequence of sending data
2. Insert `uart_rx`, `uart_tx`, and `uart` into the project as sources in order to end up with the final "uart" design. Test your design using the provided `uart_tb` testbench. Set the simulation time to 5ms. The testbench is set to "fail" when the transactions are complete to halt simulation so you don't have to worry about setting an exact simulation time. If there are errors, it will print mismatch warnings in the console.

Listing 3.1: UART Transmitter Entity

```
entity uart_tx is
port (
    clk, en, send, rst   : in std_logic;
    char                 : in std_logic_vector(7 downto 0);
    ready, tx            : out std_logic
);
end uart_tx;
```

Note: After you import the files into the project, ensure that "uart" is set as the top level design for synthesis, and "uart_tb" as the top for simulation.

2 We have the technology

2.1 Background

In order to use our UART and talk to the computer, we need to drive some output through it by sending bytes. In order to do so, we're going to create a FSM that will drive the ASCII codes for your NetID along with a control signal to the UART.

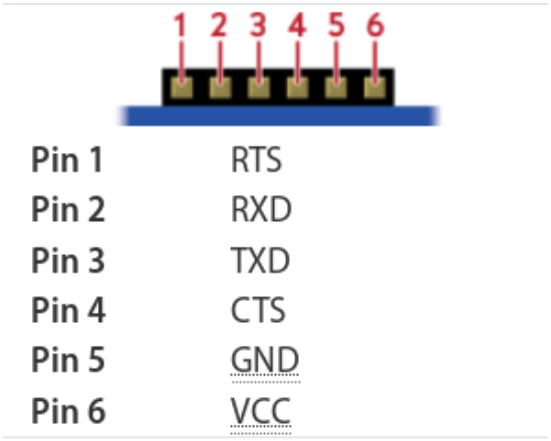
For our top level, we're also going to need to connect a USB $\leftrightarrow$ UART pmod to talk with the computer from the pins on the Zybo. In order to do so, we have to look at the physical layout of the PMOD connector on both the Zybo and the USB $\leftrightarrow$ UART adapter.

Here is the list of pins for PMOD connector JB on the Zybo, what nets in the constraints file those pins map to, and the pin-out for the adapter:

Table 3.1: Pmod JB Pin Names

JB1	T20	JB7	Y18
JB2	U20	JB8	Y19
JB3	V20	JB9	W18
JB4	W20	JB10	W19

(a) Pmod Pin Ordering Diagram [3]



(b) USB ↔ UART adapter pinout [1]

Figure 3.3

We are going to plug our adapter into the top row of PMOD plug JB on the Zybo. RXD and TXD are the `RX` and `TX` of the device. We connect them to the `TX` and `RX` (respectively) of our entity (notice the crossing!). GND and VCC are already handled by the PMOD port as pins 5 and 6 which we have no control over. CTS (clear to send) and RTS (request to send) are optional signals that will be ignored as we do not need flow control, so those pins will tied to ground in our main design.

2.2 Task

1. Create a simple entity called "sender" that takes as input a reset, a clock, a clock enable, a button, and a "ready" signal and outputs a "send" signal and an 8 bit "char", with the following behavior:
   - It contains an n-long array of 8-bit vectors called "NETID" (where n is the number of characters in your NetID) that is initialized to your NetID in ASCII as well as a binary counter `i` that initializes to **0** and can count up to at least **n**
   - It contains a single process FSM that initializes to an "idle" state and changes on the rising edge of the clock when enable is **1** with the following behavior:
     - When a reset signal is asserted, it clears all of its outputs as well as counter `i` and goes to idle synchronously
     - When `ready` is **1** and `button` is **1** and `i` is less than **n**, it should assert `send` to **1**, place

`NETID(i)` on `char`, increment `i`, and transition to a "busyA" state. If ready is **1** and button is **1** but `i` is equal to **n**, it should reset `i` to **0** and stay in idle.

- After entering "busyA", it should transition to "busyB"
- After entering "busyB", it should change send to **0** and go to "busyC"
- It should stay in state "busyC" until `ready` changes to **1** and `button` is **0**, at which point it transitions back to "idle"

2. Instantiate "sender" and "uart", a modified "clock_div" that produces a 115200 Hz output, and a pair of "debounce", in a top level design according to the diagram in figure (3.4).
3. Create a modified constraints (xdc) file to map the appropriate signals on the board to your top level design. See the background section above to see what needs to be done to properly connect the USB ↔ UART adapter.
4. Load the design onto the board and test it.

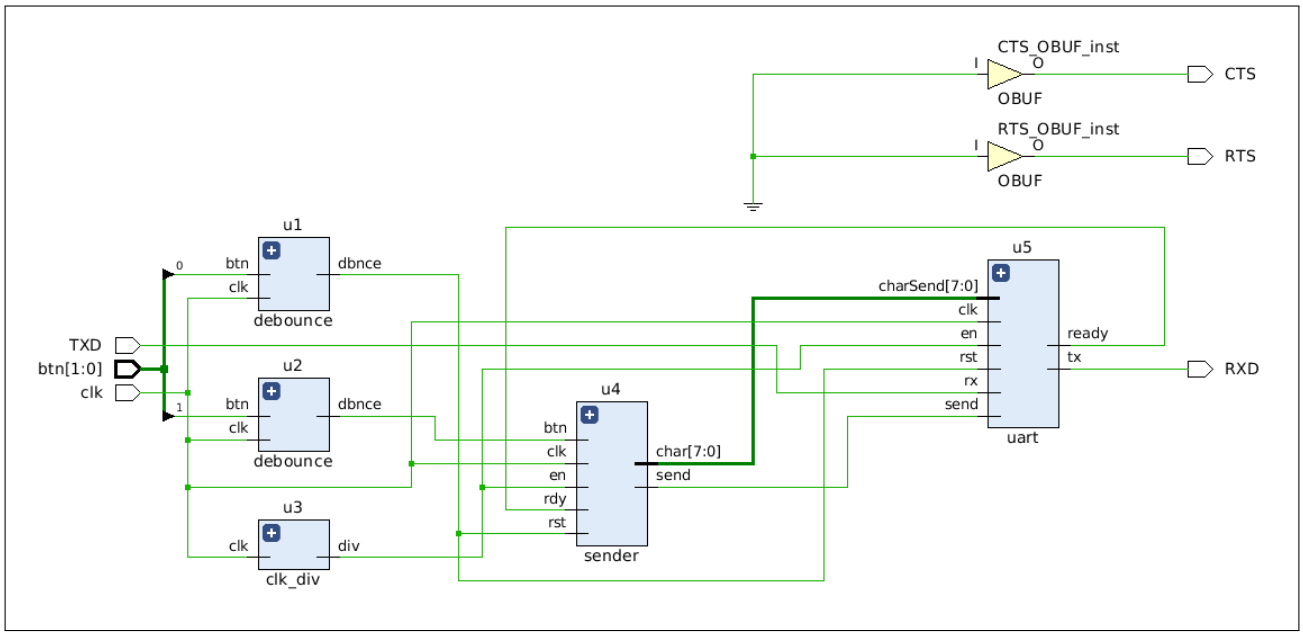


Figure 3.4: Top Level Block Diagram

2.3 Testing

In order to test your design, you are going to need to hook up the USB ↔ UART adapter to a computer and open a serial terminal. For this section, we are going to give directions for how to do so in both Windows and Linux.

In order to hook up the cable and have it properly detected, you may or may not need to install a device driver. Linux should natively support it. For Windows, open "Device Manager" and check if there is a device located under "Ports (COM & LPT)". If there is not, you need to install the drivers, which can be found here:

http://www.ftdichip.com/Drivers/VCP.htm

Once the driver has been installed, take note of the port that it says is being used (for example, COM3). You will need this port name soon.

Now we need to download and open a simple serial terminal. In Windows, Termite works well as it does not require any installation and has a simple gui. It can be found here:
https://www.compuphase.com/software_termite.htm

For Linux, we will be using Minicom as that is what is installed on the lab computers, however any serial terminal will suffice. If you're using a debian-based Linux such as Ubuntu on a personal computer and wish to install Minicom, type the following into the terminal:

```
sudo apt install minicom
```

Windows

Once you open termite, click the "settings" button and make sure you have set the settings to match those in the image below (but substitute COM3 with your port name from device manager).

Figure (3.5) shows the serial port settings window

Tip: If you are unsure which COM port to use, try connecting to one and sending some text from the Termite terminal by typing something and pressing the Enter key. If you're sending to the adapter, an LED on it will blink quickly.

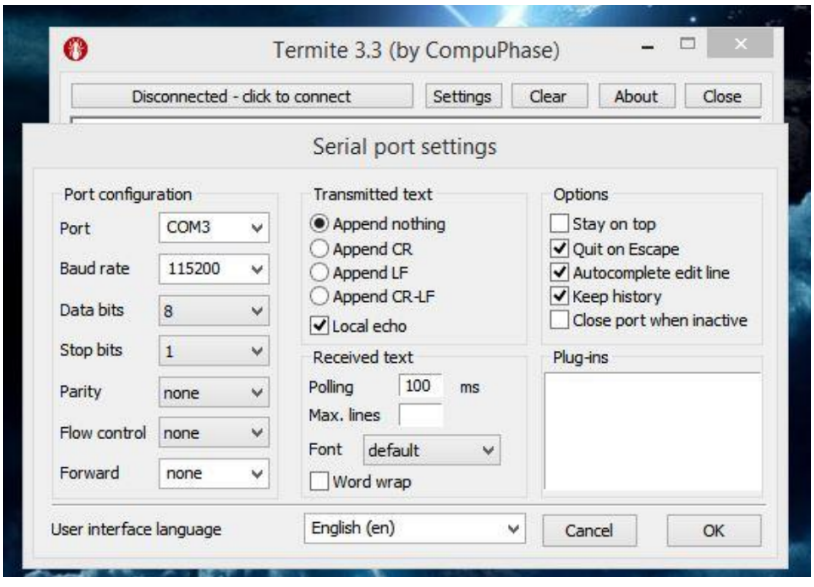


Figure 3.5: Termite Settings

Once you have input the settings, click "OK" and then click the connect button. At this point, you can

press and release the button to trigger your design to send your NetID over the serial port one character at a time and view it being received by the computer in the Termite window.

Linux

Similar to termite, you need to know which port the USB ↔ UART Pmod is plugged into to use Minicom. While there are a variety of methods you can use to determine this, one of the simpler ones is to view the connection logs via dmesg. To display and filter these logs, use the following command:

```
dmesg | grep ttyUSB
```

Once you have the port name, you can open/connect minicom by typing the following:

```
minicom -b 115200 -D /dev/device_name
```

where device_name should be of the form ttyUSBn (n = number). Alternatively, you can open minicom up into a configuration state by using:

```
minicom -s
```

and configure the settings via a terminal-gui.

3 Extra Credit

In order to properly test the receive side of the device (which has been left unused in this lab), implement a simple FSM to replace "sender" called "echo". It will receive characters from the computer and echo them back.

1. Create a simple entity called "echo" that takes as input a a clock, a clock enable, a "ready" signal, a "newChar" signal, and an 8-bit "charIn" signal. It should output a "send" signal and an 8-bit "charOut" signal, with the following behavior:
   - It contains an FSM that initializes to an "idle" state and changes on the rising edge of the clock when enable is **1** with the following behavior:
     - When `newChar` is **1** it should assert send to **1**, place `charIn` on `charOut`, and transition to a "busyA" state
     - After entering "busyA", it should transition to "busyB"
     - After entering "busyB", it should change send to **0** and transition to "busyC"
     - It should stay in state "busyB" until ready changes to **1**, at which point it transitions back

to “idle”

2. Create a properly modified constraints file
3. Create a new top level design to load the hardware onto the board (I’ll leave it up to you to figure out the block diagram) and test the echo functionality.

4 Supplied VHDL

4.1 uart_rx.vhd

Listing 3.2: UART Reciever

```
--
-- written by Gregory Leonberg
-- fall 2017
--
--------------------------------------------------------------------------------

library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;


entity uart_rx is
port (
    clk, en, rx, rst    : in std_logic;
    newChar             : out std_logic;
    char                : out std_logic_vector (7 downto 0)
);
end uart_rx;

architecture fsm of uart_rx is

    -- state type enumeration and state variable
    type state is (idle, start, data);
    signal curr : state := idle;

    -- shift register to read data in
    signal d : std_logic_vector (7 downto 0) := (others => '0');

    -- counter for data state
    signal count : std_logic_vector(2 downto 0) := (others => '0');

    -- double flop rx plus 2 extra samples to take majority vote of 3
    -- majority vote of samples helps mitigate noise on line
    signal inshift : std_logic_vector(3 downto 0) := (others => '0');
    signal maj : std_logic := '0';

begin

    -- double flop input to fix potential metastability
    -- plus 2 extra samples to take majority vote of 3 inputs (oversampling)
    -- majority vote of samples helps mitigate noise on line
    process(clk) begin
    if rising_edge(clk) then
        inshift <= inshift(2 downto 0) & rx;
    end if;
    end process;

    -- majority vote of 3 samples (oversampling)
    -- majority vote of samples helps mitigate noise on line
```

```
    process(inshift)
    begin
        if (inshift(3) = '1' and inshift(2) = '1' and inshift(1) = '1') or
           (inshift(3) = '1' and inshift(2) = '1') or
           (inshift(2) = '1' and inshift(1) = '1') or
           (inshift(3) = '1' and inshift(1) = '1') then
                maj <= '1';
        else
            maj <= '0';
        end if;
    end process;

    -- FSM process (single process implementation)
    process(clk) begin
    if rising_edge(clk) then

        -- resets the state machine and its outputs
        if rst = '1' then

            curr <= idle;
            d <= (others => '0');
            count <= (others => '0');
            newChar <= '0';

        -- usual operation
        elsif en = '1' then
            case curr is

                when idle =>
                    newChar <= '0';
                    if maj = '0' then
                        curr <= start;
                    end if;

                when start =>
                    d <= maj & d(7 downto 1);
                    count <= (others => '0');
                    curr <= data;

                when data =>
                    if unsigned(count) < 7 then
                        d <= maj & d(7 downto 1);
                        count <= std_logic_vector(unsigned(count) + 1);
                    elsif maj = '1' then
                        curr <= idle;
                        newChar <= '1';
                        char <= d;
                    else
                        curr <= idle;
                    end if;

                when others =>
                    curr <= idle;

            end case;
```

```
        end if;

    end if;
    end process;

end fsm;
```

4.2 uart.vhd

Listing 3.3: UART, Top Level

```
--
-- written by Gregory Leonberg
-- fall 2017
--
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity uart is
port (
    clk, en, send, rx, rst      : in std_logic;
    charSend                    : in std_logic_vector (7 downto 0);
    ready, tx, newChar          : out std_logic;
    charRec                     : out std_logic_vector (7 downto 0)
);
end uart;

architecture structural of uart is

    component uart_tx is
    port (
        clk, en, send, rst  : in std_logic;
        char                : in std_logic_vector (7 downto 0);
        ready, tx           : out std_logic
    );
    end component;

    component uart_rx is
    port (
        clk, en, rx, rst    : in std_logic;
        newChar             : out std_logic;
        char                : out std_logic_vector (7 downto 0)
    );
    end component;

begin

    r_x: uart_rx
    port map (
        clk => clk,
        en => en,
        rx => rx,
        rst => rst,
```

```
        newChar => newChar,
        char => charRec
    );

    t_x: uart_tx
    port map (
        clk => clk,
        en => en,
        send => send,
        rst => rst,
        char => charSend,
        ready => ready,
        tx => tx
    );

end structural;
```

4.3 uart_tb.vhd

Listing 3.4: Testbench for UART Design

```
--
-- written by Gregory Leonberg
-- fall 2017
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

entity uart_tb is
end uart_tb;

architecture tb of uart_tb is

    component uart port (
    clk, en, send, rx, rst  : in std_logic;
    charSend                : in std_logic_vector(7 downto 0);
    ready, tx, newChar      : out std_logic;
    charRec                 : out std_logic_vector(7 downto 0)
    );
    end component;

    type str is array (0 to 4) of std_logic_vector(7 downto 0);
    signal word : str := (x"48", x"65", x"6C", x"6C", x"6F");

    signal rst : std_logic := '0';
    signal clk, en, send, rx, ready, tx, newChar : std_logic := '0';
    signal charSend, charRec : std_logic_vector(7 downto 0) := (others => '0');

begin

    -- the sender UART
    dut: uart port map(
```

```
        clk => clk,
        en => en,
        send => send,
        rx => tx,
        rst => rst,
        charSend => charSend,
        ready => ready,
        tx => tx,
        newChar => newChar,
        charRec => charRec);


    -- clock process @125 MHz
    process begin
        clk <= '0';
        wait for 4 ns;
        clk <= '1';
        wait for 4 ns;
    end process;

    -- en process @ 125 MHz / 1085 = ~115200 Hz
    process begin
        en <= '0';
        wait for 8680 ns;
        en <= '1';
        wait for 8 ns;
    end process;

    -- signal stimulation process
    process begin

        rst <= '1';
        wait for 100 ns;
        rst <= '0';
        wait for 100 ns;

        for index in 0 to 4 loop
            wait until ready = '1' and en = '1';
            charSend <= word(index);
            send <= '1';
            wait for 200 ns;
            charSend <= (others => '0');
            send <= '0';
            wait until ready = '1' and en = '1' and newChar = '1';

            if charRec /= word(index) then
                report "Send/Receive MISMATCH at time: " & time'image(now) &
                lf & "expected: " &
                integer'image(to_integer(unsigned(word(index)))) &
                lf & "received: " & integer'image(to_integer(unsigned(charRec)))
                severity ERROR;
            end if;

        end loop;
```

```
        wait for 1000 ns;
        report "End of testbench" severity FAILURE;

    end process;

end tb;
```

5 Credits

Sources

[1] *Pmod USBUART*. URL: https://reference.digilentinc.com/reference/pmod/pmodusbuart/start#example_projects.

[2] *UART Frame (RS232)*. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/4/47/UART_Frame.svg/2000px-UART_Frame.svg.png.

[3] *ZyBo Reference Manual*. Feb. 27, 2017. URL: https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual.

Acknowledgments

Gregory Leonberg
Original Author
Fall 2017

Stephen DiNicolantonio
Updates/Modification and LaTeX Design
Fall 2018