Indian Institute of Information Technology Design and Manufacturing, Kancheepuram

ELE547A - System on Programmable Chip Practice

Project Report: UART Controller

Nehemiae George Roy - EVD18I018

Introduction

Communication plays a vital role in modern electronics and systems. There are billions of devices that humans interact with, and these devices must have some form of streamlined communication. If there are even a few devices that are designed to communicate differently, this could potentially lead to delays, data losses or even potential loss of human life. The varying effects of incorrect or inconsistent communication definitely depend on the scale of the systems that are working, but yet these issues can be compounded in the intersystem or intrasystem, and lead to unexpected effects.

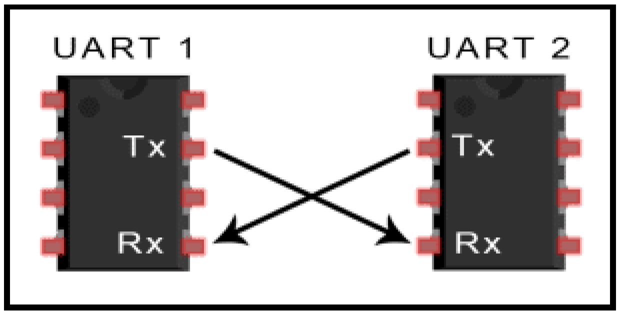
With the current market trend toward IOT, we can see the distinctive need for edge connected devices. These ‘nodes’ are usually low-powered and constrained hardware that operate to fulfill pre-programmed code as quickly as possible to reduce the latency of the system. Hence, the majority of such devices used are microcontrollers, as these provide the perfect balance between processing capabilities and power requirements. Furthermore they are easily reprogrammable whenever updates need to be rolled out. The only hindrance will be that microcontrollers are generic processors and might not be the best optimized solution for every application.

Now, in order for microcontrollers to communicate we require some protocol and pin connection that all microcontrollers must follow. The requirements of any protocol is that they provide the exact syntax or structure of the messages being transmitted, the semantics or meaning of the messages, the mode of synchronization of transmitting and receiving and the required error detection and correction methods. These requirements must clearly defined and agreed upon by the communicating devices. Furthermore, the physical pin connections must be set up appropriately for both the transmitter and receiver. There are several communication protocols for microcontrollers such as SPI, I2C, CAN, etc. and for this project we shall focus on the UART communication protocol and simulate its controller in Verilog.

In the first section, a brief literature review is given where the functioning of the different modules associated with UART is given, along with necessary requirements for any compliant device. Next, a detailed breakdown of the simulation of the UART controller in Verilog is given. A section is presented featuring the advantages and potential drawbacks of implementing a UART based communication system. Finally, details of the further scope of this project and UART applications are given.

Literature Review

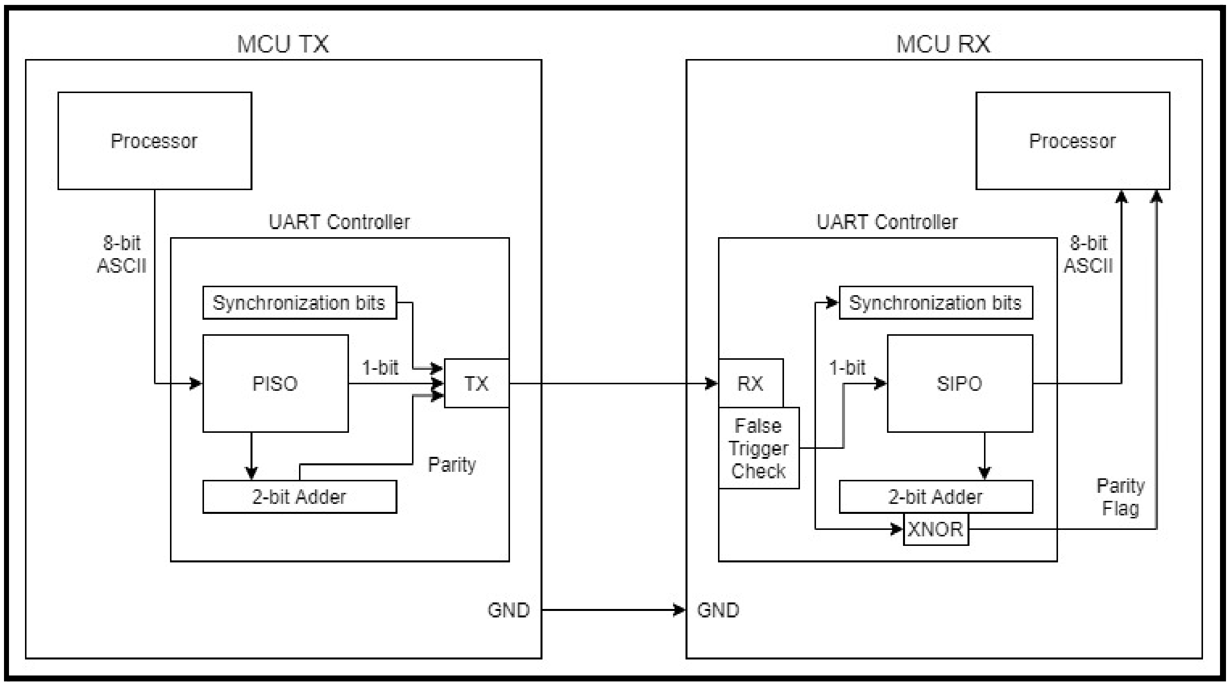
UART is an abbreviation for Universal Asynchronous Receiver Transmitter, and as the name implies this mode of communication does not require a clock signal between the devices. The only two lines that are required are the data line and reference line or ground. Since this is a serial communication protocol, bits are transferred one at a time, and the transmitter and receiver needs to be configured appropriately so that the entire word will be received correctly. This is in contrast to parallel communication, such as RAM and PCI, where multiple bus lines transmit an entire word at a time. In the latter form of communication every individual bit line of the bus must be wired and this can lead to complex circuitry. The UART controller is generally included as a peripheral on a microcontroller or other processors that can be used for serial communication to peripheral devices. The devices can be connected in either simplex, half duplex or full duplex modes. Simplex mode is when there is only transmitter and receiver, while half duplex mode allows both devices to communicate but one at a time and full duplex mode allows simultaneous communication. An important point to be noted is that the voltage pulses that are used to transmit 0s and 1s across the wire, are not generated by the UART peripheral itself, but are triggered by the processor. Earlier UART was used in the RS-232 and RS-485 systems that were 12 V and 5 V signal systems respectively, but now these ports have been decapitated.



General pin connection between UART peripherals on microcontrollers

The UART protocol does not specify device IDs and control signals, but simply sends a pre-decided length of data (usually 8 bits) to the receiver. Thus, essentially we can send the entire space of ASCII characters along the line. Hence, though we can ideally connect any amount of receivers in parallel, provided they do not require to be addressed individually, realistically we can only connect a pair of devices to avoid signal degradation from affecting the communication. The entire space of ASCII characters that can be represented by a single byte form the semantics of most UART communication.

First let us understand the general block diagram of a UART controller.



General block diagram of UART controller

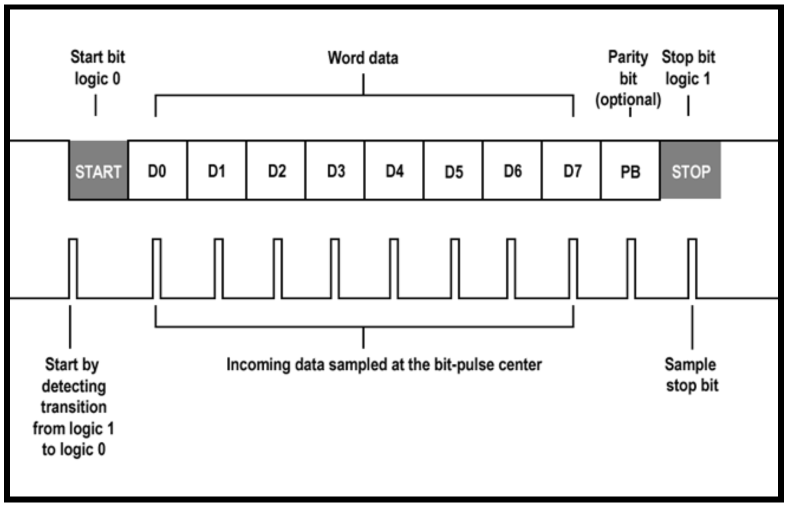
The block diagram shows the connection between two microcontrollers, with the UART controllers inside each of them. This illustration is for a simplex communication, but this is easily replicated in the reverse direction with another pair of pins. As mentioned before UART does not provide any control or acknowledgement signals and hence it is ultimately up to the processor to decide whether the microcontroller is transmitting or receiving. The processor of the transmitter sends an 8-bit ASCII word to the UART peripheral, as a byte of data. This data must be split into serial bits that can be sent across the line, and hence we use the Parallel-In Serial-Out shift register for this. At the same time, the only synchronization bits - i.e. the start and stop bits - are also included. There is an optional parity bit that can also be included to indicate whether there are an even or odd number of 1s. At the receiver end, we must take in this serial data and pack them together to form an 8-bit word. Hence, we use the Serial-In Parallel-Out shift register. From this register, we can remove the synchronization bits and the parity bit. The parity bit can be used to compare with the parity of the bits that were received during transmission and trigger a flag to the processor if necessary.

By convention, the idle state of the wire is pulled up HIGH - usually through a pull-up resistor - as this can help both UART controllers identify if the device is working properly or not and enable or disable the peripheral accordingly. The PISO and SIPO registers must be of the same size and for 8-bit data with parity check, we would require 11-bit PISO and SIPO registers. The start bit, which is a logic LOW, is sent first and the receiver checks if the first LOW received sustains for atleast half a bit period (reciprocal of baud rate). If it does not, then it is treated as a random fluctuation and is discarded to avoid false triggering. To check the parity of the data being transmitted, bitwise XOR can be performed on the transmitted and received word as this returns 1 whenever the two bits are not the same, which follows binary addition. Since there is no clock synchronization between the devices, once the start bit has been detected, the receiver will restart its internal clock to match the data being received. Then it will read the data at the middle of each clock cycle.

In order for effective communication to take place between the two devices, the UART protocol must provide a packet structure format, to identify the syntax of the communication. One of the shortcomings of the UART protocol is that it does not provide the definitions for more intelligent communication. Instead it specifies a very generic frame structure that will allow any device with a GPIO pin to be configured accordingly. This comes as a question of tradeoff between complexity and meaningful information. The various bits of frame are listed below is their order of occurrence.

1. Start Bit - This is a logic LOW signal and the receiver verifies if this signal lasts atleast half a clock cycle. Once this has been verified the receiver enables its SIPO register and the bits are shifted on every clock cycle.
2. Data Bits - This is usually 8-bits wide but is mandatory to be of the same expected size of the receiver, or else there may be not enough or too many shift registers to convert the data into parallel form.
3. Parity Bit - This an optional bit, and as always the receiver and transmitter must agree on this for correct syntax. The parity bit essentially provides an error detection mechanism in UART, but is limited in identifying only a single bit flip - an even number of bit flips or if the parity bit itself is flipped will go unidentified. Bit flips occur due to EMI because of coupling between parallel wires or a EM noisy environment, solder bridges to ground or even signal attenuation due to long data wires. The parity flag that is set by the UART can then trigger an interrupt from the receiver to transmitter to resend the data. However, in such a case full duplex communication will not be possible.
4. Stop Bit - The final bit that is transmitted is the stop bit which is a logic HIGH. The packet framing allows that there is atleast one logic level transition (when a NULL character 0x00 is transmitted) in any transmission, and if this fails then a break error is generated.

The frame structure is summarized in the below picture.



Frame Structure

Now the final component to ensure effective communication is the mode of synchronization. The UART transmitter and receiver will use internal clocks to process data through their respective shift registers, but this clock is not shared between the two devices. Unfortunately in UART the only way to perform reliable communication is to ensure that these two clocks are atleast 10% within each other. This frequency is the rate at which the data is sent across the line and is called the baud rate. Various baud rates include 9600, 14200 all the way up to 1.5 MHz. Higher frequency communication is often not possible due to constraints on signal bandwidth on the line.

Now we must note that simply because the two devices agree on a certain baud rate, does not mean that the rate of sending by the transmitter and rate of sampling by the receiver is exactly synchronized, as these both have been started at unknown points in time. Hence, when the start bit is detected, the receiver will restart the clock at that point to match with the rate of data received.

Even though we set the two controllers for a specific baud rate, the internal clocks are driven by on-board crystal oscillators that can be susceptible to slight variations from their intended frequency. As long as this deviation is within an error tolerance of 10%, the SIPO can shift the received bits within the time that the signal value on the line changes. This is proved experimentally and will be tested for the simulation being performed.

The frequency of microcontrollers vary anywhere between a few MHz to 168 MHz. This frequency is much larger than the required baud rate, and hence we require to downsample this clock signal. This is performed by the baud rate generator. Essentially, we need to find a suitable frequency divisor to bring down the clock frequency to the baud rate. An example would be a microcontroller running at 16 MHz and a baud rate of 9600 bps is desired.

$$Frequency\ Divider = \frac{16M}{9600} \approx 1667$$

Hence, with the use of PLLs, this frequency divider must be set into the system to get the appropriate baud rate. Note that the integer round-off is where slight errors can creep in.

Analog Devices has given the following equation for calculating the baud rate. This will depend on the clock configuration employed.

$$Baud\ Rate = \frac{P_{CLK}}{(M + \frac{N}{2048}) \times 2^{OSR+2} \times DIV}$$

where P_{CLK} is the main processor frequency

M and N are the fractional baud rates

OSR is the oversampling rate

DIV is the baud rate divider

The entire denominator is essentially the frequency divider of interest.

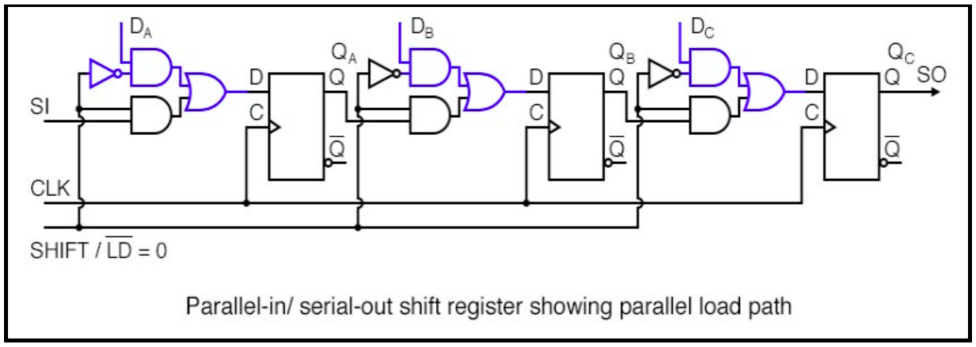
Certain error flags must also be available to check whether any unintended operation has occurred. These are listed below.

1. Overrun Error: This error occurs when the UART peripheral's SIPO register still holds the bits of one transmission cycle, while the start bit of the next transmission arrives. If there is buffer storage between the UART and the processor, this error could also occur because of insufficient buffer space to add the byte of data received. Essentially, it shows that the processor is unable to keep up with the baud rate and hence either it must be sped up or the baud rate brought down.

2. Underrun Error: This error occurs when the transmitter has sent all the available data bytes and this length is less than the expected data transmit size. Often this is not treated as an error, but additional stop bits are appended to keep the synchronization accurate.
3. Framing Error: Whenever an entire frame does not meet its specifications this error is triggered. After a start bit has been detected, the receiver will start a counter to count up to 10 cycles, when all the bits have been received. At this point, it will expect a HIGH signal to specify the stop bit and if for reason it does not receive it, an error is flagged. Erroneous data transmit sizes can lead to this error.
4. Parity Error: When the parity of byte received and the parity bit set in the MSB-1 bit of the frame, this error is set.

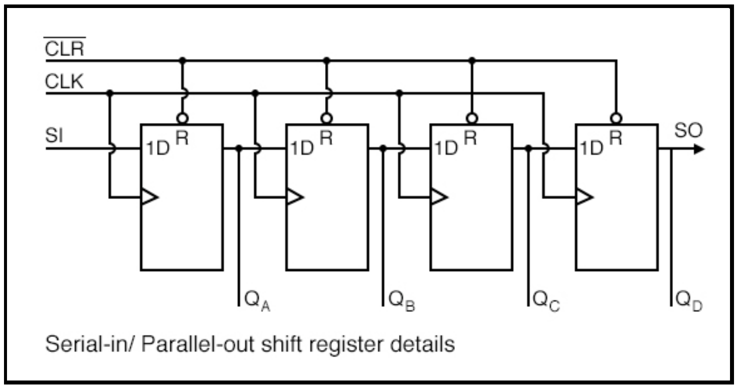
One of the disadvantages of UART is the lack of acknowledgement signals and hence any error can be detected relatively easily compared to them being corrected. If we are willing to sacrifice full duplex communication, then the receiver can transmit a request signal back to the transmitter.

The next important part are the shift registers. At the transmitter side we use the PISO register. Since we want to send the Start Bit first it will be at LSB and the Parity and Stop Bits will be the first two bits. The PISO register has a shift/load flag to choose whether we want to load the immediate bit from the processor or we want to transfer the bits across the chain. So for one cycle we choose to make it 0 to load all the bits to each flip-flop, and then next cycle onwards we make it 1, so that at every positive edge of the clock the data will shift across.


D_A
D_B
D_C
Q_A
Q_B
Q_C
SI
CLK
SHIFT / $\overline{LD}$ = 0
D
C
Q
$\overline{Q}$
SO
Parallel-in/ serial-out shift register showing parallel load path

PISO register block diagram

At the receiver end we use the SIPO register where we directly shift each bit every cycle. We must sample the data only after the all data has been shifted which is the number of stages into the time period.



SIPO register block diagram

Analog Devices has expanded their implementation of UART to include a new frame protocol. The reality is that nothing prevents us from establishing a new protocol as long as it is consistent and usable across many devices. In their frame protocol, headers are used to identify the devices and hence establishes a potential master-slave control. Based on the kind of communication command bits could be sent next - for example accessing internal registers, measured values, resetting the device, etc. The next most important and distinct difference between UART and this new implementation is that the latter supports variable data length as it sends the data length before transmitting the data. This can help improve transmission speeds for large volumes of data. Finally, trailer bits and CRC bits are appended to indicate end of transmission and a more rigorous error checking than parity.

The paper “A Review Paper on Design and Simulation of UART for Serial Communication” by Vibhu Chinmay and Shubham Sachdeva, follows a very similar approach to the described procedure to simulate the UART peripheral. In addition they used the SCCR register to set the baud rate and check the status of the transmitter and receiver. Also they used the SCSR register to check for overrun or framing errors. Timing analysis was further done to see what reliable speeds were possible.

Implementation

In this section we will now look at the different modules that build the design for the UART controller. Every module will be followed by its associated testbench, to test its proper functionality. The final testbench `UART_testbench.v` is the only point of control to the user, where we can change the value being transmitted using the `out_word` register.

Modules in Verilog act as a black box with input and output pins to communicate with the rest of the world. The `wire` type is used for all inputs to the module as these values need not be updated and hence need not be stored. On the other hand, if the output variables require to be changed in the module, then we must declare as `reg` type and then the new values can be sent to other modules.

The D flip-flop is the basic building block of both shift registers that are required for the transmitter and receiver. It is a sequential circuit element where the values depend on previous inputs. Essentially it behaves as a delay element since the values at its input is propagated to the output on every rising edge of the clock. In the following testbench, we check when the output value changes for the D flip-flop. Though we may change the input value during a clock cycle, the output is changed only in the next clock pulse since the D flip-flop is edge triggered.

```
module D_FF(D, CLK, Q);
input D, CLK;
output Q;
reg Q;

always@(posedge CLK)
begin
    Q = D;
end

endmodule
```

```
module D_FF_testbench();
reg in, CLK;
wire out;

D_FF D(in, CLK, out);

always #10 CLK = ~CLK;

initial
begin
    CLK <= 1;
    in <= 0;

    #25 in <= 1;
    #15 in <= 0;
    #20 in <= 1;
end

endmodule
```

Next we need to build the PISO shift register for the transmitter. As explained before we require the use of a combinational circuit that can toggle between loading input bits from the processor to the corresponding input pins of each flip-flop or shift the value from the previous flip-flop to the next one. For this purpose we use the `Sel` module which is level-triggered so as to synchronize with the remaining clocked elements. Since we are transmitting 7 bits (4 bit word and 3 synchronization bits), the `PISO_SR` module cascades 7 D flip-flops with 6 selection circuits alternatively. In the testbench we transmit 0000100 and every bit will be transmitted after $7 \times 20 = 140\,ns$.

```
module Sel(bit_in, D_out, SHIFT_LD, CLK, out);
input bit_in, D_out, SHIFT_LD, CLK;
output out;
reg out;

always@(CLK)
begin
    if (SHIFT_LD == 0)
```

```
        out = bit_in;
    else if (SHIFT_LD == 1)
        out = D_out;
end

endmodule
```

```
module PISO_SR(in_word, SHIFT_LD, CLK, out_bit);
input [3+3:0]in_word;
input CLK, SHIFT_LD;
output out_bit;
wire [3+3-1:0]D_out;
wire [3+3-1:0]S_out;

D_FF D6(in_word[6], CLK, D_out[5]);
Sel S5(in_word[5], D_out[5], SHIFT_LD, CLK, S_out[5]);
D_FF D5(S_out[5], CLK, D_out[4]);
Sel S4(in_word[4], D_out[4], SHIFT_LD, CLK, S_out[4]);
D_FF D4(S_out[4], CLK, D_out[3]);
Sel S3(in_word[3], D_out[3], SHIFT_LD, CLK, S_out[3]);
D_FF D3(S_out[3], CLK, D_out[2]);
Sel S2(in_word[2], D_out[2], SHIFT_LD, CLK, S_out[2]);
D_FF D2(S_out[2], CLK, D_out[1]);
Sel S1(in_word[1], D_out[1], SHIFT_LD, CLK, S_out[1]);
D_FF D1(S_out[1], CLK, D_out[0]);
Sel S0(in_word[0], D_out[0], SHIFT_LD, CLK, S_out[0]);
D_FF D0(S_out[0], CLK, out_bit);

endmodule
```

```
module PISO_SR_testbench();
reg [3+3:0]out_word;
reg SHIFT_LD, CLK;
wire in_bit;

PISO_SR PS1(out_word, SHIFT_LD, CLK, in_bit);

always #10 CLK = ~CLK;

initial
begin
```

```
    out_word <= 7'b0000100;
    SHIFT_LD <= 0;
    CLK <= 1;

    #20 SHIFT_LD <= 1;
end

endmodule
```

Similarly, we need to build the SIPO shift register for the receiver cascading 7 D flip-flops in series. The testbench transmits 7 bits that will be received as the word 0000101 after $7 \times 20 = 140\,ns$.

```
module SIPO_SR(in_bit, CLK, out_word);
input in_bit, CLK;
output [3+3:0]out_word;

D_FF D6(in_bit, CLK, out_word[6]);
D_FF D5(out_word[6], CLK, out_word[5]);
D_FF D4(out_word[5], CLK, out_word[4]);
D_FF D3(out_word[4], CLK, out_word[3]);
D_FF D2(out_word[3], CLK, out_word[2]);
D_FF D1(out_word[2], CLK, out_word[1]);
D_FF D0(out_word[1], CLK, out_word[0]);

endmodule
```

```
module SIPO_SR_testbench();
reg out_bit, CLK;
wire [3:0]in_word;

SIPO_SR SP1(out_bit, CLK, in_word);

always #10 CLK = ~CLK;

initial
begin
    CLK <= 1;

        out_bit <= 0;
    #20 out_bit <= 0;
    #20 out_bit <= 0;
    #20 out_bit <= 0;
    #20 out_bit <= 1;
    #20 out_bit <= 0;
    #20 out_bit <= 1;
end

endmodule
```

Now, we must configure the transmitter module that will be responsible for controlling the state of the line in all conditions. The data ready signal will be set by the test bench once data is ready. Since the line is HIGH at idle state, 7 bits of 1s are used to fill up the D flip-flops. Once data is ready we include the synchronization bits and parity flag, and then we set the shift registers to load mode. The `time_flag` and `clock_count` counts how many cycles have elapsed from this point onwards. After one clock cycle the PISO is changed to the shift mode. Also after 7 cycles, the wire must be reset so that the next cycle it will be pulled high. And after 8 cycles, we set the transmit flag and reset the time flag. From the test bench at this point, the data ready is made 0, so we go back to the first `if` block.

```
module Processor_TX(in_word, data_ready, CLK, tx_flag, out_bit);
input [3:0]in_word;
input data_ready, CLK;
output tx_flag, out_bit;
reg tx_flag = 0;
reg [3+3:0]data = 7'b1111111;
reg SHIFT_LD = 0;
integer time_flag = 0, clock_count = -1;

PISO_SR PS1(data, SHIFT_LD, CLK, out_bit);

always@(posedge CLK)
begin
    if (data_ready == 0)
        begin
        data = 7'b1111111;
        end
    else if ((data_ready == 1) && (time_flag == 0))
        begin
            data[0] = 0;
            data[4:1] = in_word;
            data[5] = in_word[0] ^ in_word[1] ^ in_word[2] ^ in_word[3];
            data[6] = 1;
            //data[5] = 0; //Parity Flag Trigger
            //data[6] = 0; //Break Flag Trigger
            SHIFT_LD = 0;
            time_flag = 1;
            clock_count = -1;
        end

     if (time_flag == 1)
        begin
           clock_count = clock_count + 1;
           if (clock_count == 1)
                SHIFT_LD = 1;
           else if (clock_count == 7)
                begin
                    data = 7'b1111111;
                    SHIFT_LD = 0;
                end
           else if (clock_count == 8)
                begin
                    tx_flag = 1;
                    time_flag = 0;
                end
        end
end

endmodule
```

Next at the receiver module, the input bit is continually passed from the transmitter to the D flip-flops. The moment a start bit is received, `time_flag` and `clock_count` are set. Then after 7 clock cycles, first the receive flag is set and then the break condition and parity is checked. If there are no issues the required word is extracted out. Then the `time_flag` is reset.

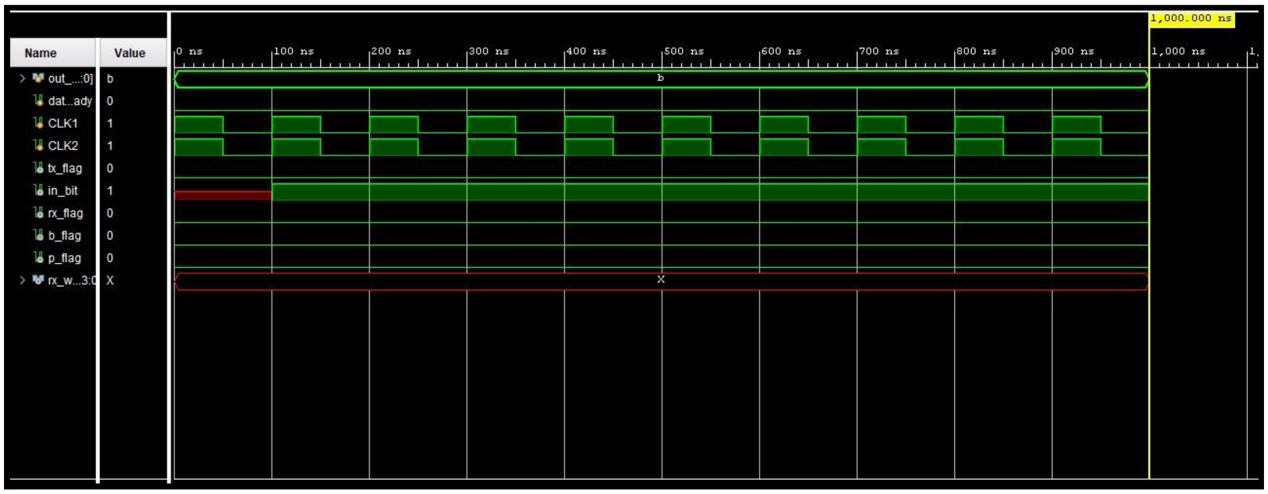
```
module Processor_RX(in_bit, CLK, rx_flag, b_flag, p_flag, data);
```

```
input in_bit, CLK;
output rx_flag, b_flag, p_flag;
output [3:0]data;
reg [3:0]data;
reg rx_flag = 0, b_flag = 0, p_flag = 0;
wire [3+3:0]out_word;
integer time_flag = 0, clock_count = -1;

SIPO_SR SP1(in_bit, CLK, out_word);

always@(posedge CLK)
begin
    if ((in_bit == 0) && (time_flag == 0))
        begin
        time_flag = 1;
        clock_count = -1;
        end

    if (time_flag == 1)
        begin
           clock_count = clock_count + 1;
           if (clock_count == 7)
                begin
                    rx_flag = 1;
                    if (out_word[6] == 0)
                        b_flag = 1;
                    else if (out_word[4] ^ out_word[3] ^ out_word[2] ^ 
out_word[1] != out_word[5])
                        p_flag = 1;
                    else
                        data = out_word[4:1];
                    time_flag = 0;
                end
        end
end

endmodule
```

Finally, the testbench basically connects both modules, feeding the `in_bit` of `Processor_TX` to `Processor_RX`. As a sample value we take the 1011 or 0x0B value.

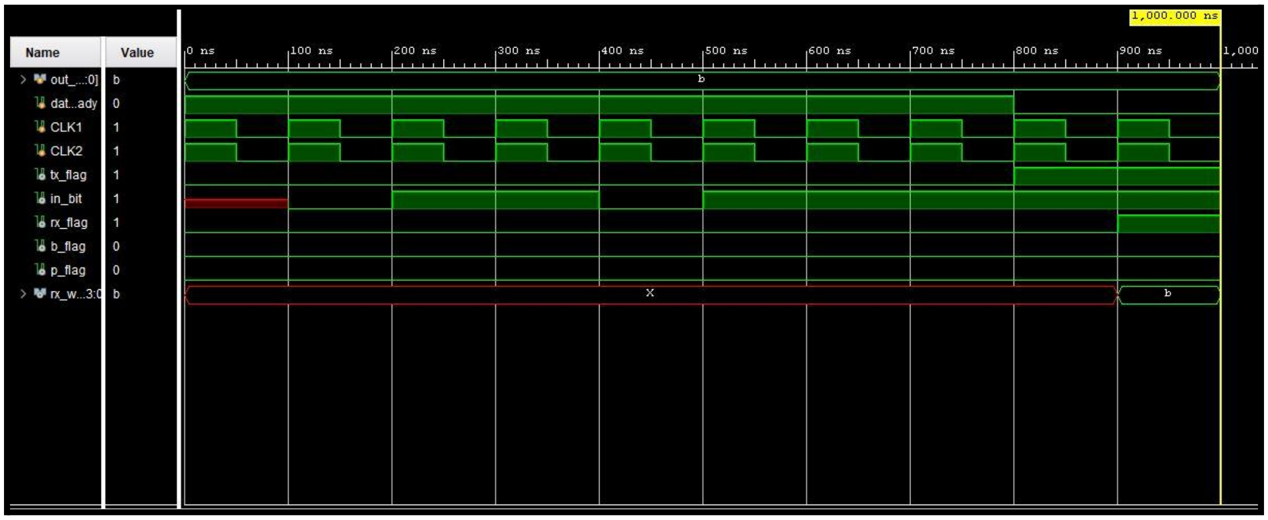
```
module UART_testbench();
reg [3:0]out_word;
reg data_ready, CLK1, CLK2;
wire tx_flag, in_bit;
wire rx_flag, b_flag, p_flag;
wire [3:0]rx_word;

Processor_TX P1(out_word, data_ready, CLK1, tx_flag, in_bit);
Processor_RX P2(in_bit, CLK2, rx_flag, b_flag, p_flag, rx_word);

always #50 CLK1 = ~CLK1;
always #50 CLK2 = ~CLK2;

initial
begin
    out_word = 4'b1011;
    data_ready = 1;
    CLK1 <= 1;
    CLK2 <= 1;

    #(100 * 8) data_ready <= 0;
end

endmodule
```

Let us now look at the output graphs that we obtain for some test cases. At the idle state, `data_ready` is always 0. After a clock cycle, `in_bit` is HIGH.



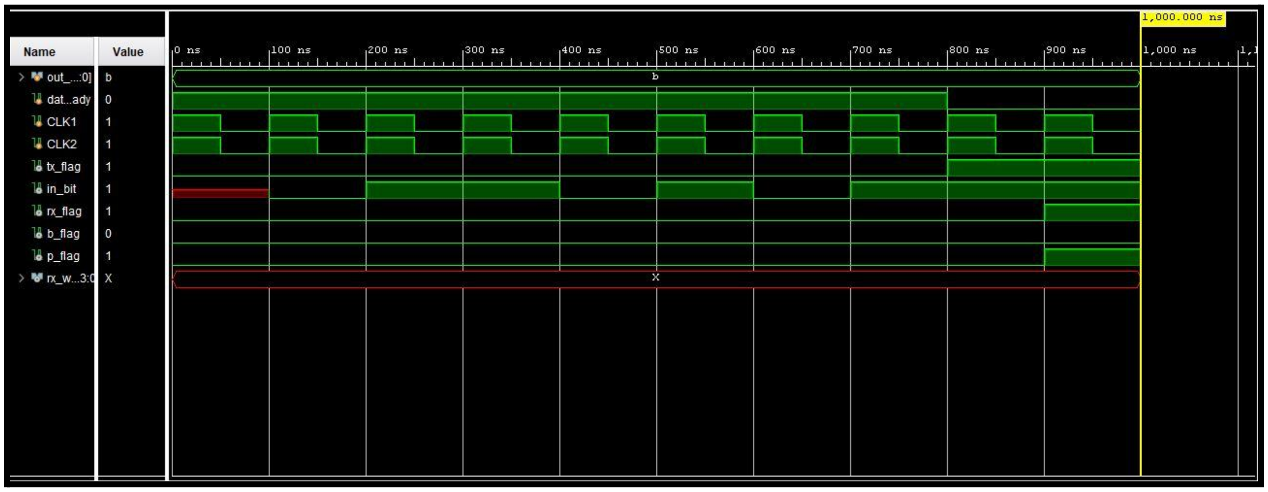
Output during idle state

Next when transferring the B value, the `data_ready` will be HIGH for 8 cycles. The `in_bit` shows 0, which is the stop bit. Then there is 1101 which is the reverse of 1011 since we are transferring LSB first. Then the parity flag 1 and then the stop bit 1. After this transmit flag goes HIGH. The `rx_flag` and `rx_word` will be delayed by one cycle and we get B.



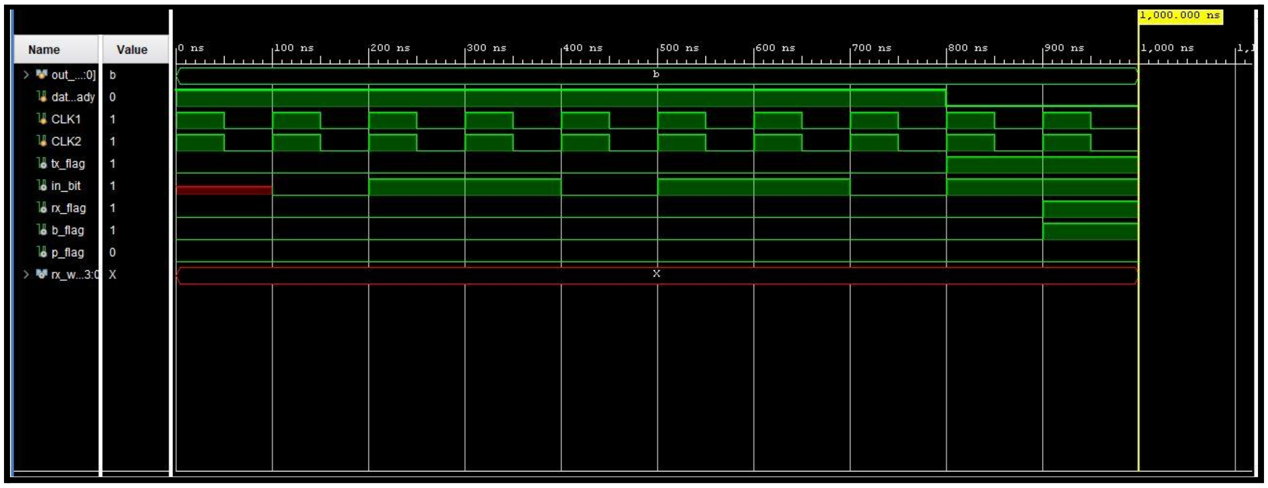
Output during data transmission of 0x0B

Next, when giving the same input but parity flag as zero we obtain 0110101. When the `rx_flag` goes HIGH, the `p_flag` is raised and we have no data.



Output during data transmission of 0x0B with parity error

Finally, we send a 0 instead of a 1 for the stop bit and similarly the `b_flag` goes HIGH.



Output during data transmission of 0x0B with break error

Multiple trial-and-error simulations were performed to note the maximum allowed frequency deviation. The ability to detect an error actually depends on the data being transmitted, as the only way to detect an error is with parity and break flag. For this example upto 16% deviation there was no error. For good measure, the two clocks must be within 10% of each other.

Processor TX Time Period	Processor RX Time Period	Frequency Deviation	Status
100	100	0%	Correct
100	90	10%	Correct
100	88	12%	Correct
100	86	14%	Correct
100	84	16%	Correct
100	82	18%	Parity Error
100	80	20%	Incorrect Value

Pros and Cons

UART is by today's standards a very inefficient mode of communication, but it does pose certain advantages over modern communication protocols. The most important of these are the simplicity and ease of setting up a transmitter and receiver modules. Only the data line and ground line are the required physical connections between the two devices and this constraint is especially important for microcontrollers with limited GPIO pins. While we do sacrifice transmission speed compared to parallel communication, we gain in limited physical connections. Furthermore the non-requirement of a clock signal is advantageous in small applications as long connection lines can heavily distort the clock signal's integrity and it is a source and sink for EMI. Hence, serial communication that requires a clock - like I2C - must repeatedly be tested with oscilloscopes to ensure the reliability of the signal.

The disadvantages of UART are plentiful and the most troublesome aspect is that the programmer must ensure that the two internal clocks are within 10% error tolerance and if it is not unrelated errors are flagged that can lead to confusion, however with the increasing reliability of crystal oscillators this issue is minimized. Furthermore the use of parity bit for error detection is limited as it can only detect a single bit flip. Also, if any kind of error correction mechanism is employed then we cannot perform full duplex communication. The general standard for UART is limited and does not address a device specifically, and hence we can only connect one receiver and the data length of transmission is fixed which may not be always efficient. However, as explained previously if the frame structure is modified (like Analog Devices have done) we can overcome this issue.

Further Scope and Conclusion

One disadvantage in the simulation performed is that we need to wait for an entire transmission to complete before we can transmit or read again. To avoid this we can use buffers at both ends between the UART peripheral and the processor, so that we can keep data ready and improve transmit time. However if at receiver we do not remove the data quickly enough then overrun error will happen and hence the processor clock and baud rate must be properly configured. Also, the simulation that has been done is completely dependent on the processor to send data to and read data from the UART modules, and this can lead to CPU overloading where the CPU is blocked for a long time and cannot process other threads. So to avoid this we use the Direct Memory Access (DMA) peripheral, which bypasses the processor and directly transmits the data to memory. In this method we can employ double buffering, where we can be receiving data while simultaneously transmitting. The DMA will be connected to the UART across a high speed bus and this will help lower any overrun errors.

Some applications of UART include direct transfer to a serial terminal module where we can display ASCII characters. Or we can connect a microcontroller to our computer via a UART to USB converter and emulate a virtual COM port using the USB CDC API on the microcontroller. Then we can use serial terminals like Putty or even MATLAB to read the data.

In conclusion, we have studied the UART controller and simulated it functioning to transmit a word of data between two devices. The possible error conditions were looked into and the associated flags configured accordingly. Observations were made on the possible frequency deviation of the internal clocks of the two devices and the limitations of using UART communication. By simply mirroring the modules between the transmitter and receiver we can extend this application to half-duplex and full-duplex mode.

References

1. “A Review Paper on Design and Simulation of UART for Serial Communication”, Vibhu Chinmay, Shubham Sachdeva, IJIRT
2. “BASICS OF UART COMMUNICATION”, https://www.circuitbasics.com/basics-uart-communication/

3. “UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter”, https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html
4. “Universal Asynchronous Receiver-Transmitter”, https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
5. “Shift Registers: Serial-in, Parallel-out (SIPO) Conversion”, https://www.allaboutcircuits.com/textbook/digital/chpt-12/serial-in-parallel-out-shift-register/
6. “Shift Registers: Parallel-in, Serial-out (PISO) Conversion”, https://www.allaboutcircuits.com/textbook/digital/chpt-12/parallel-in-serial-out-shift-register/