UART Bit-Clock Generation Using Assertion Statements

Ben Cohen http://SystemVerilog.us

The DUT: UART transmitter

A *UART* is a Universal Asynchronous Receiver Transmitter device utilizing an *RS232* serial protocol. A typical *UART* consists of a transmitter partition and a receiver partition. A *CPU* typically loads an eight-bit word into a *UART*. The *UART* frames the data word and parity (if any) with a *START* bit (a logical 0) at the beginning, and a *STOP* bit (a logical 1) at the end of the word. It sends the framing information along with the data and parity in a serial manner from the Least Significant data Bit (*LSB*) to the Most Significant Bit (*MSB*), followed by the parity bit. Figure 1 represents the timing waveform of a *UART* message issued by a *UART* transmitter.

The serial data is received by a *UART* receiver. Synchronization is based on the negative transition of the *START* bit that resets a divide-by-16 counter clocked by a clock 16 times the bit-clock. This counter is then used to create mid-clock when it reaches a value of 7; it is then used to clock the data stream. The receive *UART* stores the serial data into a receive shift register. When all the data is framed, it alerts the receive *CPU* that data is ready (*rdy* signal). The *rxdata* signal represents the received 8-bit word.

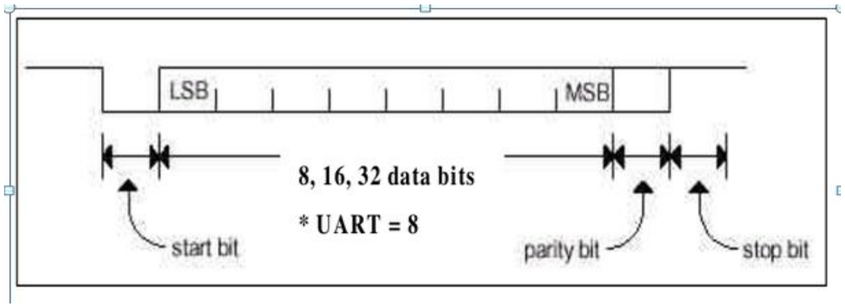

Figure 2 Interface format of a UART serial data

Synchronized bit-clock generation

The falling edge of the START bit is used to synchronize the clock division setup to the start of a bit. It also indicated the start of a new message. Thus, any previous clock division process needs to be reset. The `cover property cp_bit_clk` and the `cp_reset`, and supporting functions provide the generation of the bit-clock, as shown below. In this model, `rxd` (serial data input) is connected to the `serial_out` of the transmitter. The (**$fell**(rxd, @(**posedge** clk16x)) || !rst_n) cancels any on-going cover statement in progress. Since there is a need to trigger the bit-clock generation right after a (**$fell**(rxd, @(**posedge** clk16x)) the `rxd` signal is delayed by one-bit and is clocked at the 16x clock; the delayed signal is called `rxd_r`, which is very close in time to the synchronization signal. In this `cover property`, the **$fell**(rxd_r) is used as an antecedent to start the generation of the bit-clock. However, since every bit of the serial data does not necessarily generate a falling edge at bit-time, it is necessary to generate a bit-synchronous signal to restart this process at every bit time. For this, the `restart` signal is used as that flag. The `restart` signal is reset or set using the `set_restart` function called from a sequence match item. In addition the bit-clock is set or reset with the `set_bit_clk` function. The initialization of the `restart` and the awaiting for a new message is performed in the `cp_reset cover property` statement. The reset of a new message is performed in the `p_data` property when a new message is detected.

```
// generate the bit clock
    cp_bit_clk: cover property(
        @ (posedge clk16x)
        disable iff ($fell(rxd, @(posedge clk16x)) || !rst_n)
        (($fell(rxd_r) || restart), set_bit_clk(1), set_restart(0)) |->
        ##7 (1, set_bit_clk(0)) ##8
        (1, set_bit_clk(1), set_restart(1)));

    cp_reset: cover property(  // actions at reset time
        @ (posedge clk16x)
        disable iff (0)
        (!rst_n, set_restart(1), set_new_msg(1)) );

  always @ (posedge clk16x) begin  rxd_r <= rxd;  end
  function void set_bit_clk(bit k);   bit_clk=k;   endfunction : set_bit_clk
  function void set_restart(bit k);   restart=k;   endfunction : set_restart
  function void set_new_msg(bit x);   new_msg    =x; endfunction : set_new_msg
```

The above bit-clock generation can be performed in classes, using an RTL-like style, as follows:

```
  class mon_uartrx;
    virtual interface uart_if vif;
    // bit clk16x,
    bit rst_f;
    bit rxd;
    bit rdy;
    bit clk1x;

    bit[10:0] rxdata_r;   // the receive  register
    bit[3:0] count16_r; //  for divide by 16
    bit rxmt_r;   //  Receive register empty
    bit rxd_r;   //  registered serial input
    bit parity_enb =1'b1, odd_parity, parity_err;


    task xmt_tsk;
        forever begin
            @ (posedge vif.clk16x) begin : Rx_Lbl
                rxd_r    <= vif.serial_out;
                //--   reset
                if (vif.rst_n == 1'b0) begin
                    count16_r        <= 4'b0000;     //  reset divide by 16 counter
                    rxmt_r           <= 1'b1;            // new message starting
                end

                //   new bit start
                else if (rxmt_r && rxd_r == 1'b0) begin
                    count16_r        <= 4'b0000;     // reset divide by 16 counter
                    rxmt_r           <= 1'b0;            // transmit reg not empty
                end
                // if @ 16X clock rollover
                else if(count16_r == 4'b1111)
                    count16_r        <= 0;

                //  Normal count16 counter increment
                else
                    count16_r        <= count16_r + 1'b1;

            end  :Rx_Lbl
        end
    endtask :xmt_tsk
```

All test code can be downloaded from http://SystemVerilog.us/uart_vh.tar

Caveat: The point of this paper is to show the capabilities of SVA, and as such, SVA was used for almost every aspect of the data gathering and supporting constructs, such as the generation of the bit-clok from the 16x clock. A user may want to have a combination of constructs, e.g., RTL-like and SVA.