UART (VHDL)

Created by Scott Larson, last modified by Robert Nelson on Sep 08, 2017

Code Download

Version 1.0: uart.vhd

Initial Public Release

Features

- VHDL source code of a Universal Asynchronous Receiver/Transmitter (UART) component
- Full duplex
- Configurable baud rate
- Configurable data width
- Configurable parity (even/odd/none)
- Configurable oversampling rate for receive data
- No flow control

Introduction

This details a UART component for use in CPLDs and FPGAs, written in VHDL. The component was designed using Quartus II, version 13.1.0. Resource requirements depend on the implementation. Figure 1 illustrates a typical example of the UART integrated into a system.

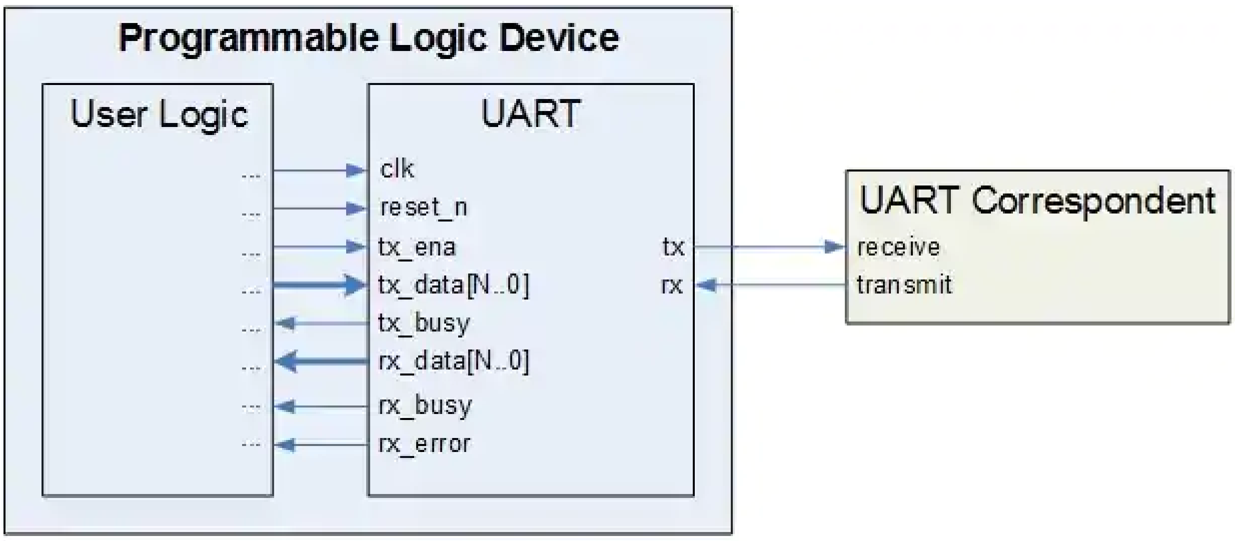

Figure 1. Example Implementation

Background

A UART is a device used for asynchronous serial communication. It consists of two lines for data transmission, RX and TX, one in each direction. Sometimes additional lines are included to implement flow control, most commonly RTS (Ready to Send) and CTS (Clear to Send). The transmission speed, data width, parity, and flow control are all configurable and must be set the same for both UART correspondents.

Figure 2 show the data framing for a transmission. Absent communication, the line is held high to indicate that it and the transmitter are not damaged. A transaction begins with a low start bit. The data word follows. Next comes an optional parity bit, which can be configured to even parity, odd parity, or no parity. Finally, a high stop bit ends the transaction.

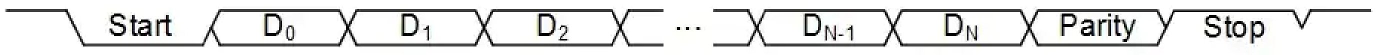

Figure 2. Data Framing

The communication can be simplex, half duplex, or full duplex. In this component, the transmit and receive lines operate independently. While this makes the component inherently full duplex, it can also operate as either simplex or half duplex if controlled by the user logic to do so.

Theory of Operation

Generating Baud Rate and Oversampling Rate Clock Enables

The baud rate is the transmission speed of the data in bits/second. The oversampling rate is the number of times the receive circuitry samples the receive input per baud period (i.e. per data bit).

This component achieves the baud rate and oversampling rate by generating clock enable pulses at those frequencies. These signals are derived from the frequency of the system clock *clk*, which must be specified in the generic parameter *clk_freq*.

A counter within the component generates a baud pulse that occurs at the baud rate. This periodic pulse enables the system clock to operate the transmit circuitry at the baud rate. There is a small error introduced each baud period, since the generated baud rate must be an integer multiple of the system clock. However, this error will not exceed one period of the system clock.

Similarly, another counter generates an oversampling pulse that occurs at a frequency = oversampling rate * baud rate. This pulse enables the system clock to operate the receive circuitry at the oversampling rate. Note that this counter is also reset at each baud pulse, so that counting errors do not accumulate beyond one baud period. In this manner, the achieved baud rate of the receive circuitry is identical to the transmit circuitry baud rate.

Transmit Circuitry

When the *tx_ena* input is asserted, the data on *tx_data* is latched into an internal shift register. At this time, the parity bit is also calculated using XOR logic and latched into the same shift register, along with the start and stop bits. The baud pulse then periodically enables the system clock to shift out the register contents to the *tx* line at the baud rate. The *tx_busy* output indicates to the user logic when the transmission is complete and the circuitry is ready to accept new data to send.

Receive Circuitry

The receive circuitry monitors the *rx* input on each oversampling pulse. If it detects a logic low, it starts counting and recognizes an incoming start bit once it detects a sufficient number of consecutive low inputs. At this point, it begins shifting the value of the *rx* line into a shift register at the baud rate, achieved by counting oversampling pulses.

Once the entire data word is shifted into the register, the receive circuitry verifies the data parity using XOR logic. It then outputs the received data on the *rx_data* port and flags any error detected on the *rx_error* port. A high-to-low transition on the *rx_busy* port signifies to the user logic that new receive data is now available.

Configuring the UART

The UART is configured by setting the GENERIC parameters in the ENTITY. Table 1 describes the parameters.

Table 1. Generic Parameter Descriptions

Generic	Data Type	Default	Description
clk_freq	integer	50_000_000	Frequency of the system clock input (PORT clk) (Hertz)
baud_rate	integer	19_200	Data link baud rate (bits/second)
os_rate	integer	16	Oversampling rate to find the center of receive bits (samples per baud period)
d_width	integer	8	Data width (bits)
parity	integer	1	0: no parity bit (parity_eo is irrelevant) 1: include parity bit
parity_eo	standard logic	'0'	'0': even parity '1': odd parity

Port Descriptions

Table 2 describes the UART's ports.

Table 2. Port Descriptions

Port	Width	Mode	Data Type	Interface	Description
clk	1	in	standard logic	user logic	System clock
reset_n	1	in	standard logic	user logic	Asynchronous active low reset
tx	1	out	standard logic	UART correspondent	Transmit output
tx_ena	1	in	standard logic	user logic	H: latches in tx_data and initiates a transmission L: no transmission initiated
tx_data	N*	in	standard logic vector	user logic	Data to transmit
tx_busy	1	out	standard logic	user logic	H: indicates a transmission is in progress, new tx_ena requests are ignored L: no transmission is in progress and clear to request another transmission
rx	1	in	standard logic	UART correspondent	Receive input
rx_data	N*	out	standard logic vector	user logic	Last data received
rx_busy	1	out	standard logic	user logic	H: new reception is in progress L: reception has completed and the resulting rx_data and rx_error are available to read
rx_error	1	out	standard logic	user logic	H: a start bit, parity bit, or stop bit error was detected during the last receive, so received data on rx_data may not be valid L: no error detected during the last receive

Notes
* N is the specified data width, set by the *d_width* generic

Transactions

Although part of the same component, transmit and receive transactions don't affect one another and therefore operate independently.

Transmit

Figure 3 shows the timing diagram of an example transmission. Note that the *baud_tb* signal shown is internal to the UART and is included here only for purposes of explanation. The user does not need to consider this signal to use the UART.

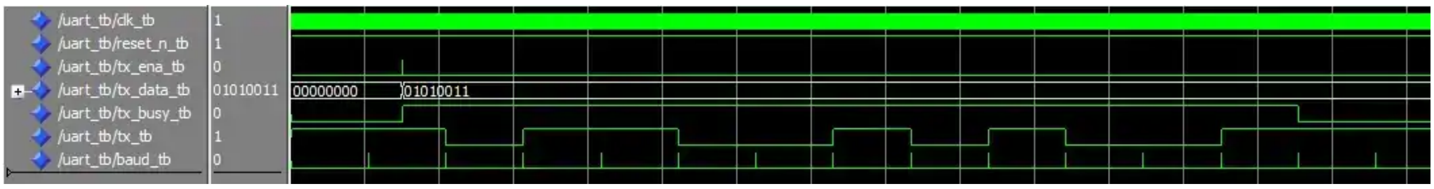

Figure 3. Example Transmit Timing Diagram

A low logic level on the *tx_busy* output port indicates that the component is ready to send data. The component latches in the data presented on the *tx_data* port on the first rising edge of the *clk* where the *tx_ena* input is asserted. On the following *clk*, the component asserts the *tx_busy* signal to indicate that the UART transmit circuit is busy. Any inputs to the transmit circuitry during this condition are ignored. The actual transmission begins once the next baud period pulse *baud_tb* is generated within the UART. At this point, the *tx* signal outputs the logic low start bit and the transmission is underway. The UART transmits the least significant bit (LSB) of *tx_data* first. In the transaction shown, an even parity bit follows the data word. Once the logic high stop bit is sent and the transmission is complete, the *tx_busy* signal deasserts, indicating that the transmit circuitry is ready for the next transaction.

Receive

Figure 4 depicts the timing diagram of an example reception. A receive transaction is initiated by the UART's correspondent via a logic low start bit at the *rx* input port. Once the UART detects that the *rx* input line has been low for half of a baud period, it recognizes the start bit and asserts the *rx_busy* signal to notify the user logic that a receive transaction is in progress. It proceeds to shift in the bits at the predetermined baud rate, calculate and verify the parity bit, and ensure that a logic high stop bit occurs at the correct time. Once it detects the stop bit, it outputs the received data to the *rx_data* port, with the first bit received as the LSB. It also outputs the *rx_error* signal to indicate whether or not a start bit, parity bit, or stop bit error was detected. It simultaneously deasserts the *rx_busy* signal to inform the user logic that the receive transaction is complete and that the results are available on the *rx_data* and *rx_error* ports.

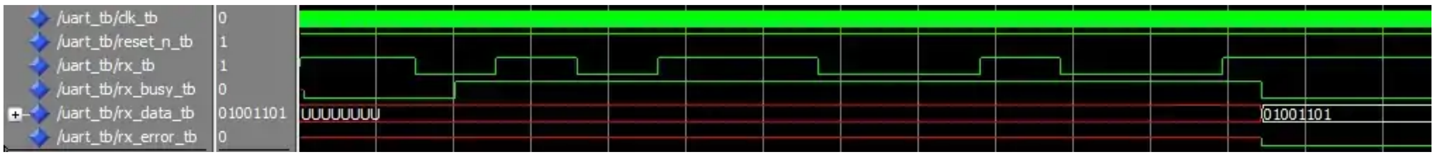

Figure 4. Example Receive Timing Diagram

Figure 5 shows how the UART detects incoming bits on the receive line. The UART monitors the *rx* input at each oversampling pulse. In this simulation, the oversampling rate (*os_rate*) is set to 16, so 16 samples occur per baud period. Once the UART detects that the *rx* input is logic low for 8 consecutive samples (*os_rate*/2), it determines that a start bit is present. It then shifts in a data bit every 16th sample, which corresponds to the middle of each receive bit. The higher the oversampling rate, the closer to the middle of the bit this is guaranteed to be.

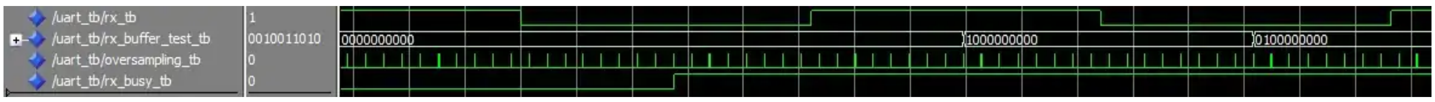

Figure 5. Receive Bit Reception Within the UART

Reset

The *reset_n* input port must have a logic high for the UART component to operate. A low signal on this port asynchronously resets the component. During reset, the component holds the *tx_busy* port high to indicate that it is not available to send data. Any transmit currently underway is discontinued, and the *tx* output assumes a logic high state. The *rx_busy* port is held low to indicate that no receive is in progress. Any receive currently in progress is abandoned, and the *rx_data* and *rx_error* output ports clear. Once released from reset, the *tx_busy* port deasserts on the following *clk*, indicating the UART's readiness to communicate.

Conclusion

This UART is a configurable programmable logic component that accommodates communication through a simple asynchronous serial interface. It allows a user to specify the system clock, baud rate, data length, parity scheme, and oversampling rate.

Contact

Comments, feedback, and questions can be sent to eewiki@digikey.com.

No labels

trk