Transmitter & Receiver (TX/RX)

Specification Document v1.00

Overview

This document specifies the interface and behavior of the Transmitter (TX) and Receiver (RX) modules for the ECE551 design project. Each TX and RX pair is responsible for unidirectional serial communication of data between routers.

Each TX & RX unit is attached to an associated Router Core, with four pairs per core. For the purpose of designing the TX and RX, you do not need to know the inner workings of the Router Cores. All communication with the Cores and between the TX and RX are governed by protocols described in this document. If your TX and RX units conform to those protocols, they will work properly with the Router Core unit and with each other.

A diagram showing the interconnection between the TX, RX, and Router Core is given in Fig. 1.

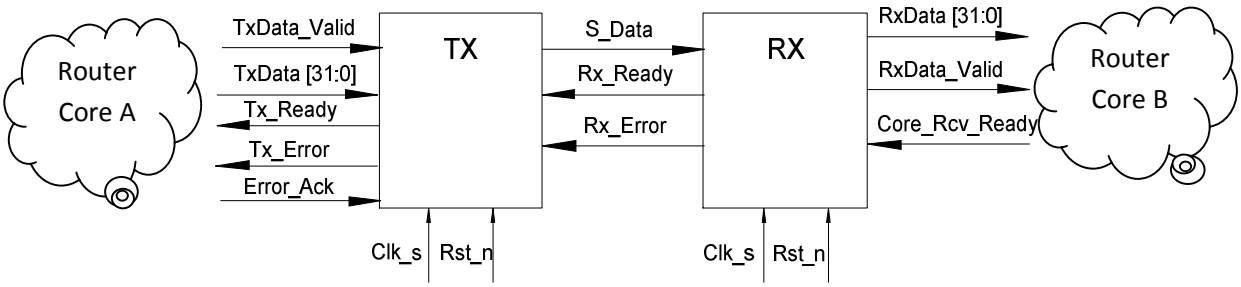

Fig 1: TX/RX Connections

The TX and RX are each composed of one or more state machines that control the operations of the unit and one or more buffers to store packets before those packets are sent or after they are received. A single buffer will probably provide the simplest approach and use the least area. Multiple buffers may be more complex and use more area, but could lead to higher throughput performance, since you will be able to load packets while transmission is taking place.

Transmitter Interface

Your TX unit should use the following interface. Please name your signals using the same names to ensure that your module will work properly with the testbenches used for grading. More detailed descriptions of the behavior of these signals are given later in this document.

Global Interface

input Clk_s – The global TX/RX clock. This clock is synchronized between transmitter and receiver. The Router Core does not necessarily use the same clock.

input Rst_n – The global asynchronous active-low reset signal.

TX / Router Core Interface

input TxData_Valid – This signal serves as part of the TX/Core handshake. When TxData_Valid is high, it means that the input TxData has valid data available on it.

input [31:0] TxData – This signal represents a 32-bit packet for transmission.

input Error_Ack – This signal acknowledges that the Core has seen that Tx_Error was asserted.

output Tx_Ready – This signal serves as part of the TX/Core handshake. When Tx_Ready is high, it means that the TX is ready to accept a packet from the Core.

output Tx_Error – This signal is used to tell the Core that an error occurred while transmitting a packet on this link.

TX/RX Interface (TX side)

output S_Data – The serial data channel.

input Rx_Ready – This signal tells the transmitter it can begin sending data on the serial channel.

input Rx_Error – This signal is asserted for one cycle when the parity check on a received packet fails.

Receiver Interface

Your RX unit should use the following interface. Please name your signals using the same names to ensure that your module will work properly with the testbenches used for grading. More detailed descriptions of the behavior of these signals are given later in this document.

Global Interface

input Clk_s – The global TX/RX clock. This clock is synchronized between transmitter and receiver. The Router Core does not necessarily use the same clock.

input Rst_n – The global asynchronous active-low reset signal.

RX / Router Core Interface

input Core_Rcv_Ready – This signal serves as part of the RX/Core handshake. When Core_Rcv_Ready is high, it means that the Core is capable of accepting a packet from the receiver.

output RxData_Valid – This signal serves as part of the RX/Core handshake. When RxData_Valid is high, it means that the input RxData has valid data available on it.

output [31:0] RxData – This signal represents a 32-bit packet which has been received from the transmitter, de-serialized, and is now presented to the Core.

TX/RX Interface (RX side)

input S_Data – The serial data channel.

output Rx_Ready – This signal tells the transmitter it can begin sending data on the serial channel.

output Rx_Error – This signal is asserted for one cycle when the parity check on a received packet fails.

Handshaking

The TX and RX operate on a single clock, **Clk_s**. The Router Core, however, will operate on a different clock that may run at a different speed from the TX/RX clock. Therefore, any signals on the interface between the TX & Core or the RX & Core are said to be *crossing clock domains*.

When signals cross clock domains it is no longer possible for their behavior to be described easily using a clock, since an operation that takes one cycle in a certain clock domain may take many cycles in another domain. This becomes an issue when data needs to be passed between domains, because it is important for the sender of the data to know when the receiver has finished reading the data, otherwise the sender might start changing the data before the receiver has had enough time to read it. *Note that the term 'receiver' is being used in a generic sense in this section, rather than referring to the RX unit*. To overcome this problem, signals that cross clock domains need to operate using a handshaking protocol.

In this project, handshaking is generally done using two signals: a *valid* signal from the sender indicates to the receiver that there is data it needs to read. A *ready* signal from the receiver indicates to the sender that the receiver is capable of reading new data. The handshaking process goes as follows (assume that both READY and VALID start low after reset):

- When the receiver has a buffer available to load new data it raises the READY signal.
- If the sender sees the READY signal is high *and* the sender has data it needs to send, it raises the VALID signal.
- Once the receiver has loaded the data from the sender (usually into a register operating on the receiver's clock domain), it lowers the READY signal.
- When the sender sees that the READY signal is low, it lowers the VALID signal.

Under no circumstances should you change the order of the handshake. To make sure you avoid any potential pitfalls in handshaking, you need to enforce strict ordering of the handshake process. Make sure you follow these rules:

- **Never** raise READY unless VALID is low.
- **Never** drop READY unless VALID is high.
- **Never** raise VALID unless READY is high.
- **Never** drop VALID unless READY is low.

A waveform showing a generic handshaking exchange is shown in Fig. 2.

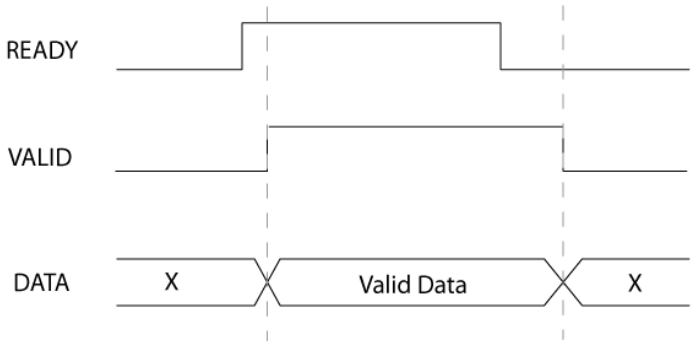

Fig. 2: Generic handshaking example

The TX and RX units take part in three different handshake protocols.

TxData Handshake

This handshake takes place between the TX and Router Core, and is used to load packets from the Core into the TX. The Core acts as sender using the **TxData_Valid** and **TxData** signals, and the TX acts as the receiver using the **Tx_Ready** signal. The TX should wait until it has a buffer free to load a packet before initiating the handshake by raising **Tx_Ready**.

RxData Handshake

This handshake takes place between the RX and Router Core, and is used to load packets from the RX into the Router Core. The RX acts as sender using the **RxData_Valid** and **RxData** signals, and the Core acts as the receiver using the **Core_Rcv_Ready** signal. The Core should wait until it has a buffer free to load the packet from the RX before initiating the handshake by raising **Core_Rcv_Ready**.

TxError Handshake

This handshake also takes place between the TX and Router Core, and is used to acknowledge that the Core has received the parity error signal that originally comes from the RX unit. The process works as follows:

1. A parity error is detected in the RX. The RX raises the **Rx_Error** signal for one cycle of **Clk_s**.
2. The TX sees the **Rx_Error** signal and raises **Tx_Error** high on the next clock edge.
3. When the Core sees that **Tx_Error** is high, it raises **Error_ACK**.
4. When the TX sees that **Error_ACK** is high, it lowers **Tx_Error**.
5. When the Core sees that **Tx_Error** has been lowered, it lowers **Error_ACK**.

Transmission Protocol

The TX and RX operate on a synchronous clock (**Clk_s**), and the sending and receiving of data is governed by the transmission protocol, which will be described here. The interface between TX and RX consists of three signals: **Rx_Ready** and **Rx_Error** (driven by the RX) and **S_Data** (driven by the TX).

1. After reset, assume that the TX and RX both start in their idle states. In the idle state, the RX should keep the **Rx_Ready** and **Rx_Error** signals low. The value of **S_Data** is considered "Don't care".
2. When the RX has buffer space available to receive a packet from the TX, it should raise the **Rx_Ready** signal high.
3. Once **Rx_Ready** is high, the TX can begin transmission as soon as it has a packet ready to send. To initiate the transfer, the TX sends a special *start sequence* on the **S_Data** line. The start sequence consists of five consecutive 0's.
4. Serial transmission of the packet starts on the next cycle after the start sequence and continues until the entire packet and associated parity bit have been sent. The first bit sent should be the most significant bit of the packet. The last bit sent should be the parity bit.
5. After receiving each packet, the RX computes the parity and compares it to the parity bit sent by the TX.
    a. If parity matches, the RX keeps the packet and eventually passes it on to the Router Core (see the RxData Handshake section).
    b. If the parity fails, the RX "drops" the packet – that is, it does not pass the data on to the Router Core. The RX raises **Rx_Error** high for one cycle of **Clk_s**.
6. If the RX is immediately able to store another packet, it may leave **Rx_Ready** high and go back to Step 2; otherwise it should drop **Rx_Ready** low and go back to Step 1.

A waveform showing the relative timing of various transmission protocol signals is shown in Fig. 3. An example of a successful transmission is shown in Fig. 4. An example of a transmission with parity error is shown in Fig. 5.

Your TX and RX units will be tested separately, so they must conform to this transmission protocol to be compatible with the testbenches used for grading. You should not alter the protocol in any way.

.

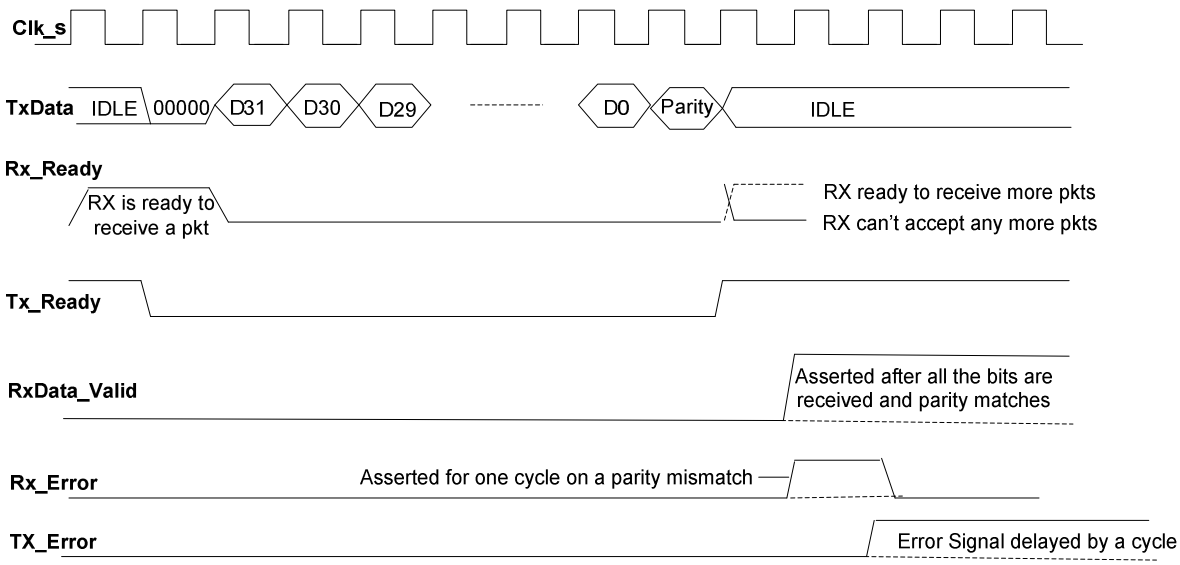

Fig. 3: Relative timing of various signals involved in the Transmission Protocol.

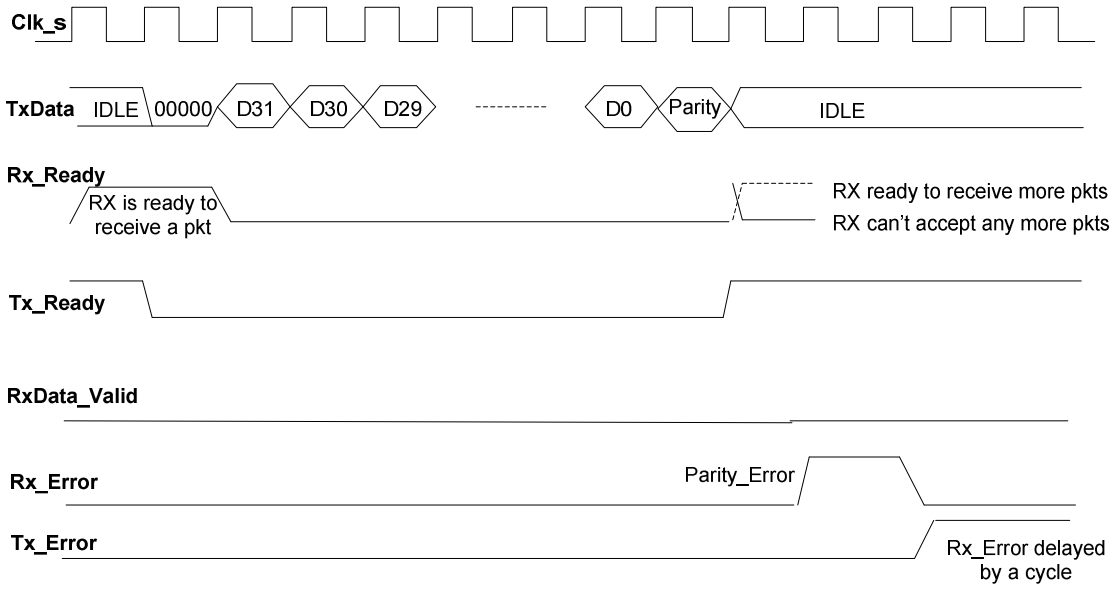

Fig. 4: Waveform demonstrating transmission with a parity error.

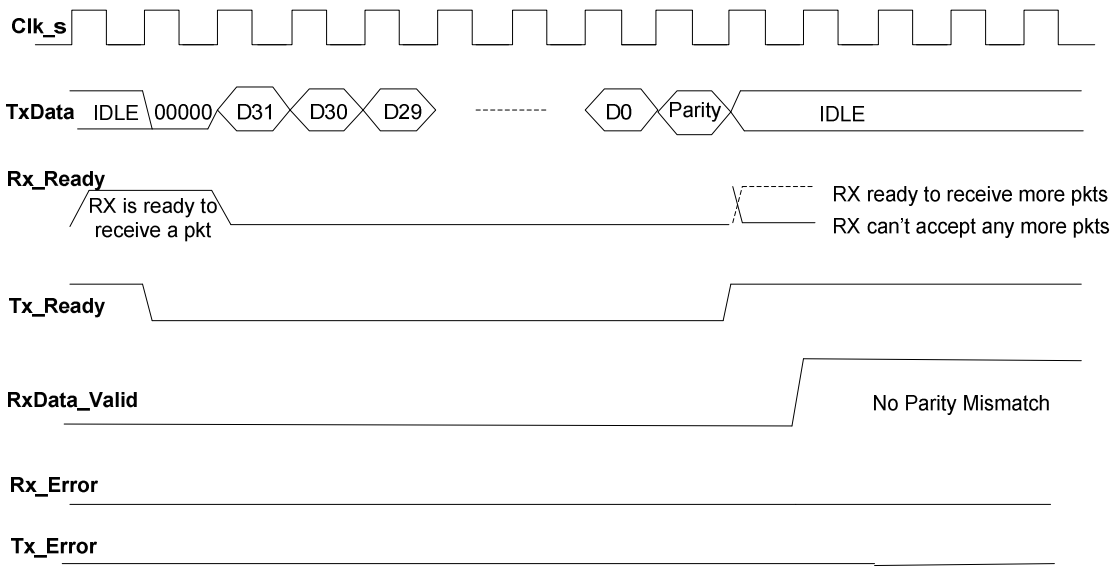

Fig. 5: Waveform demonstrating transmission without parity error. Note that the timing of the RxData_Valid signal depends on the RX/Core handshake, as described in the Handshake section.

Parity

The TX and RX utilize a one-bit ***odd parity*** to detect errors during transmission. An odd parity bit is computed in the TX and is transmitted one cycle after the 32-bit packet. The parity calculation is based on the original packet and does *not* include any bits that were inserted due to Bitstuffing.

Bitstuffing

Because the sequence 00000 has a special identity as the start sequence in our transmission protocol, a sequence of five zeros must **never** appear on the **S_Data** line, except when being used as a start sequence. This means that we must use a special encoding if a group of five zeros appears in our packet data + parity bit.

The encoding we will use is called *bitstuffing*. In the bitstuffing protocol, any time we see four consecutive zeros appear in our bitstream when transmitting a packet, we insert a one as the next bit. We do this ***regardless of whether the next bit is a one or a zero.*** Therefore, we also stuff a one into the bitstream any time there are four consecutive zeros, other than during the start sequence.

On the receiver end, any time we see four consecutive zeros, *except when we are waiting for the start sequence*, the receiver should discard the next bit.

Because we do not know in advance how many bits must be stuffed into each packet, the number of bits sent in each transmission is variable. Each transmission message will contain at least 33 bits (32-bit packet + parity), but may contain additional stuffed bits depending on the contents of the transmission. This is illustrated in Fig. 6. Because the message size is variable, the receiver must keep track of the number of bits it has received and not discarded to know when a transmission has ended.

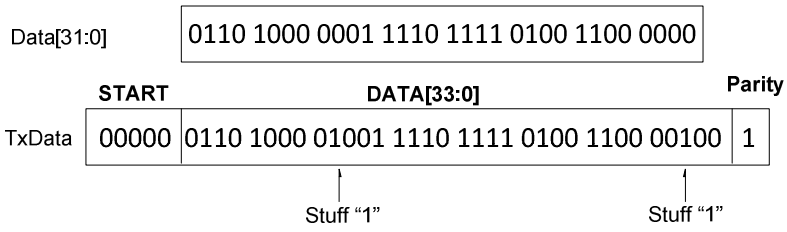

Fig. 6: Bitstuffing in the serial bitstream.

Design Tips

- **Read this document thoroughly and follow it to the letter.** The easiest way to lose points is to leave out a required feature or deviate from the specified protocols.
- **Spend some time planning your design before you start coding**. Draw out the hardware you will need and make state diagrams for any state machine. You can include these in your project report later.
- **Start Simple.** If you have a choice of implementing something in a simple way and a more complex way, do the simple implementation first. Once you have tested it and seen it working, you can add complexity piece by piece and monitor changes in the test results. This is much easier than starting out with a very complex design and trying to debug it.

- **Divide and conquer**. Rather than trying to make one large state machine, start with smaller state machines that handle different functions. For example, each of the handshaking protocols in the TX can be implemented as a separate state machine. It is easier to debug two small state machines than one large one.
- **Exploit design diversity**. It's often hard to find bugs when testing your own code, especially when you don't realize you've made a mistake in an interface. If you are designing the TX, have your partner design the RX or the TX testbench independently. If you have made mistakes in implementing the transmission protocol or handshaking, you are more likely to discover them. Remember, you will not get to see the testbenches used for grading in advance, so it's important that you follow the specifications exactly.
- **It's never too early to start thinking about synthesis**. When you are planning your design, try drawing out the hardware that would be synthesized. Once you've finished writing the module, load it up in the synthesis tool and see if you get any errors. Running Analyze and Check Design in DesignVision can quickly uncover certain types of mistakes that might take a long time to find in ModelSim.

Document History

This document will be updated as necessary to address frequently-asked questions and fix any errors. The latest version of the document will always be available on the course website. You can find a summary of the version history here.

V1.00 – Initial release

Authors: T. Gregerson, V. Nalamalapu