Application Note

An Impulse C Compatible Stream Interface for the National DS92LV16 Serializer/Deserializer

Scott Bekker
Ross K. Snider, Ph.D.
Signal Processing and Neural Instrumentation Laboratory
Electrical and Computer Engineering
Montana State University, Bozeman

Overview

This document describes the VHDL code written to implement an Impulse C compatible stream over a National DS92LV16 serializer/deserializer serial link. The VHDL describes a process to perform handshaking to establish a connection and a means of data transfer. Once the connection is established, data can be transferred in either direction with the standard ImpulseC stream interface.

Introduction

The Impulse C stream is a means by which two Impulse C processes can communicate through a standard interface. In order to achieve communication between processes on different FPGA boards, it was necessary to implement an Impulse C compatible stream that is able to communicate over the DS92LV16 serial links. The steam that was created is not as general as the standard Impulse C stream, as there are limitations such as data width imposed by the hardware. The serializer hardware allows a maximum of sixteen bits to be sent during each clock cycle. Since it is also necessary to send control flags such as FIFO status, write enable, and end of stream signals in addition to the raw data, not all of the sixteen transceiver bits are available for data transmission. Thus three of the sixteen bits are used for control flags and the rest are available for data with widths up to thirteen bits. Another difference between this stream definition and the Impulse C stream is the need to initialize the serial link before using it. A handshaking scheme is implemented to ensure both transceivers are locked to the serial data stream before allowing data transmission. In the following sections, the details of the handshaking process as well as the specifics of data transmission are discussed.

Initializing the serial connection

The stream implementation utilizes a simple handshaking process to ensure that both the remote and the local hardware transceivers are locked onto the serial data stream prior to proceeding with data transfer. Figure 1 shows the control flow for the handshaking procedure. The first step is sending out a synchronization pattern while waiting for the local transceiver to lock to the serial data stream. Once locked for a period of a few tens of milliseconds, to allow for switch bounce effects (e.g. if a cable was pulled off and replaced), the state machine alternately sends out flags and the synchronization pattern for 240 clock cycles each. The synchronization pattern allows the remote DS92LV16 to lock onto the data stream if it has not done so already. The ser-des must receive the synchronization pattern for 150 clock cycles to ensure synchronization with the serial data stream. Therefore, the slightly larger value of 240 clock cycles was chosen for the synchronization period to make sure the remote receiver is able to lock to the serial stream. The transmitted flags indicate the status of the local state-machine allowing the remote state-machine to know its state. If a valid flag response is received, the first acknowledge state is entered which simply waits for the remote transceiver to reach the same state. When the remote state-machine reaches the first acknowledge state, the local state-machine progresses to the second acknowledge state. In the second acknowledge state, a sequence of flags is sent out for a period until it is determined that the remote state-machine is also in this state or until a timeout occurs. Upon leaving the second acknowledge state, the connected state is entered signifying that both the remote and local transceivers are locked and data transmission can begin. If the local lock signal is ever lost, e.g. due to disconnecting the serial cable, the state machine reverts to the initial state and the handshaking process is repeated.

Handshaking State Machine Description (see Appendix and Figure 1)

send_sync state (lines 95 – 110):

In the send_sync state the sync_out bit is set high. This tells the transceiver to send a synchronization pattern allowing the remote transceiver to lock to the serial stream. The state machine remains in this state until the counter variable reaches 0x200000 signifying the local transceiver has been locked to the serial stream for 26 ms. This value was arbitrarily chosen, but does work with this particular hardware. After being locked for 26 ms, send_sync_240 state is entered.

send_sync_240 (lines 112 – 135) and send_flags_240 (lines 137 – 161) states:

In send_sync_240 the sync_out bit is held high to enable sending the synchronization pattern. The variable flag_cnt is incremented each clock cycle. When flag_cnt reaches 240, the state changes to send_flags_240 and flag_cnt is set to zero. In send_flags_240 the sync_out bit is cleared and a flag sequence is sent out instead of the synchronization pattern. In both of these states, the data received from the local transceiver is monitored. The counter variable is used to keep track of the number of clock cycles specific flags were received. If the counter reaches 0x000FFF in either of these two states, the local state machine will advance to the send_ack1 state. However, if the local transceiver loses synchronization with the serial data stream in either state, the state machine will revert back to the send_sync state.

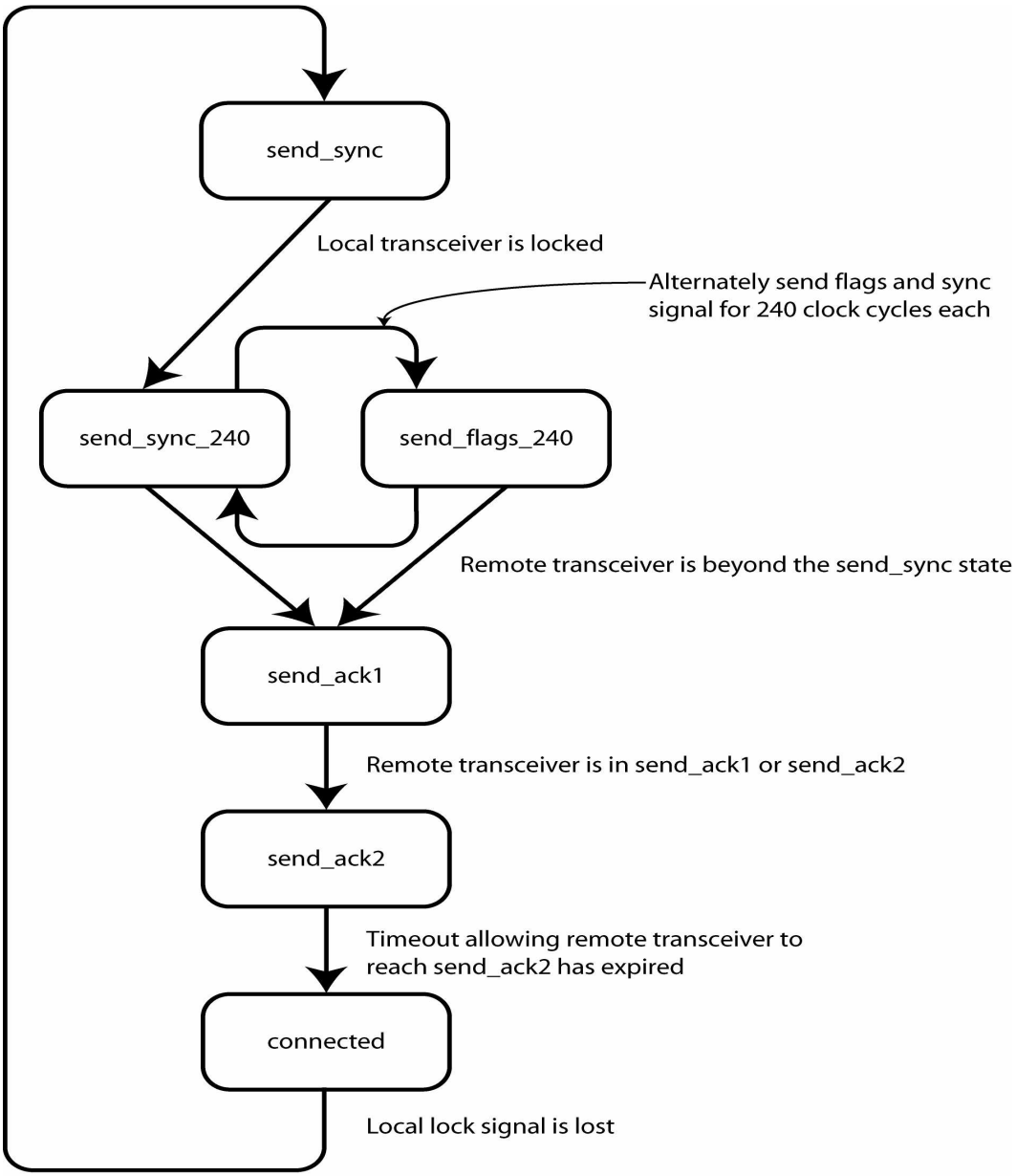

Figure 1. Handshaking State Machine Diagram

send_ack1 state (lines 163 – 180):

The send_ack1 state is used to let the remote state machine know that it has advanced beyond the previous two states. This is done by sending out new different flag sequence corresponding to this state. The local state machine remains in this state until the remote state machine has either reached this state or an advanced state for 240 clock cycles as determined by the value in flag_cnt. Upon receiving the appropriate flags for 240 clock cycles, the send_ack2 state is entered.

send_ack2 state (lines 182 – 203):

Once this state is reached, both ends of the serial connection know that the opposite end has established a connection. This state is simply to allow the remote state machine to exit the send_ack1 state. Flags are sent out until it has been determined that the remote state machine has been in this same state for 240 clock cycles or until a timeout occurs. When either scenario happens, the connected state is entered.

connected state (lines 206 – 215):

In the connected state, the link_established bit is held high indicating to the other processes that the serial connection is available for their use. The data sent to the transceiver is set to data_to_transceiver which is controlled by the data transmission logic. The state machine remains in this state unless the local transceiver loses synchronization with the serial data stream in which case the send_sync state is entered and the handshaking process is repeated.

Data Transmission

Upon initializing the serial connection, it is now possible to send data. As mentioned previously, up to thirteen of the sixteen bits can be used for data transmission. The lower bits were arbitrarily chosen for this purpose while the high bits are used for flags. The data-width can be specified during instantiation by means of a VHDL generic. The following diagram illustrates the packet sent to the serializer.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIFO Too Full	Write Enable	End Of Stream	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data

A feedback mechanism is implemented to stop the remote sender from sending data if the local FIFO is too full. If the local FIFO is too full, the “FIFO too full” flag is set in the outgoing packet. The remote transceiver will then read this flag and stop sending data. This process works in both directions as the serial links are bidirectional. The “Write Enable” and “End of Stream” flags are associated with the data in the corresponding packet. These flags let the receiver know if the data are valid and if the stream is finished.

The interface to this VHDL stream is identical to the standard ImpulseC stream plus hardware transceiver interface. The VHDL stream must be instantiated for each hardware transceiver that is to be used for an ImpulseC stream. Each stream has an interface for a producer and a consumer. The data sent into the producer port will appear on the consumer port of the remote stream instantiation and vice versa. Figure 2 illustrates a typical stream setup.

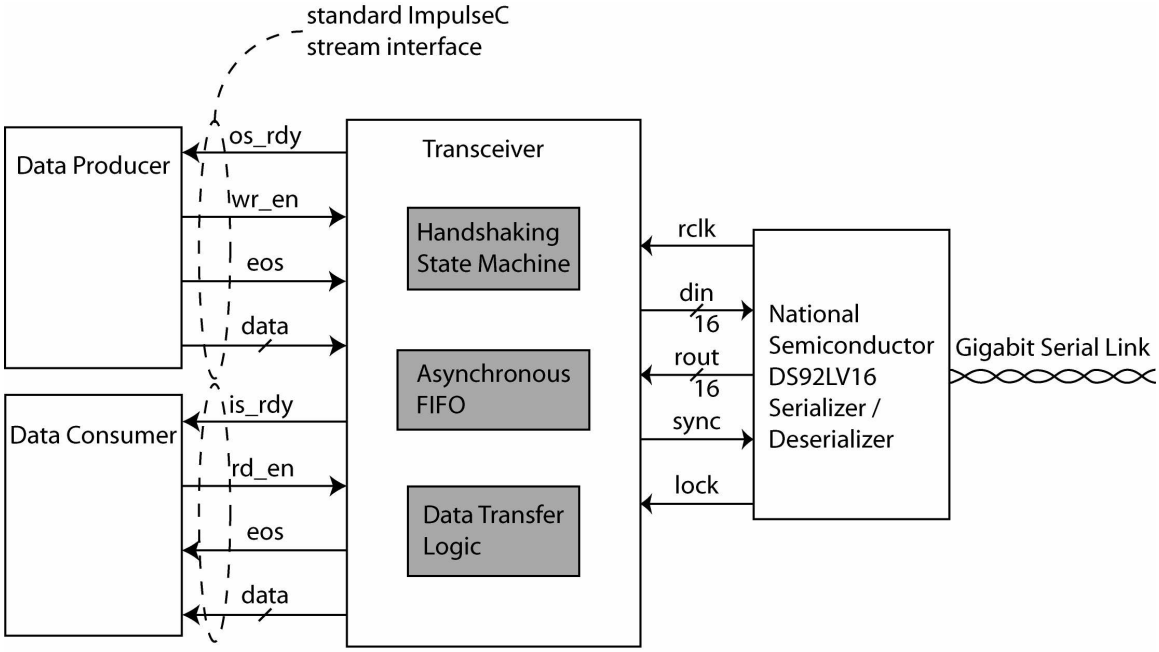

Figure 2. Stream to serdes interface

Concluding remarks

The stream implementation described above can only be used for data widths up to thirteen bits. If it is necessary to send data consisting of more than thirteen bits, the data can first be broken down into smaller portions directly in the Impulse C program and the pieces can be sent sequentially. A drawback of the above implementation is that, depending on the word size, several of the bits sent across the serial link may never contain any data, which reduces potential bandwidth. If a more sophisticated method were used to send the data such as consecutively sending several 16-bit data words followed by 16 bits of flags, higher data throughput could be achieved.

Appendix – VHDL Code Listing

```
--transceiver.vhdl
--This VHDL code describes the implementation of an eight-bit ImpulseC
--compatible stream for use over the DS92LV16 serializer/deserializer.
--
--Scott Bekker
--6/24/04

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity transceiver is
    generic(data_width : natural := 8); -- Max data width: 13 due to
                                        -- 16 bit serdes bus width - 3 flag bits used
    port ( clk : in std_logic;
           -- hardware transceiver ports
           rclk : in std_logic;
           din : out std_logic_vector(15 downto 0);
           rout : in std_logic_vector(15 downto 0);
           sync_out : out std_logic;
           local_lock_active_low : in std_logic;

           --producer ports
           producer_os_rdy : out std_logic;
           producer_wr_en : in std_logic;
           producer_eos : in std_logic;
           producer_data_in : in std_ulogic_vector(data_width - 1 downto 0);

           --consumer ports
           consumer_os_rdy : out std_logic;
           consumer_rd_en : in std_logic;
           consumer_eos : out std_logic;
           consumer_data_out : out std_ulogic_vector(data_width - 1 downto 0));

end transceiver;

architecture Behavioral of transceiver is

component async_fifo IS
    port ( din: IN std_logic_VECTOR(15 downto 0);
           wr_en: IN std_logic;
           wr_clk: IN std_logic;
           rd_en: IN std_logic;
           rd_clk: IN std_logic;
           ainit: IN std_logic;
           dout: OUT std_logic_VECTOR(15 downto 0);
           full: OUT std_logic;
           empty: OUT std_logic;
           rd_count: OUT std_logic_VECTOR(6 downto 0));
END component;

TYPE state_type is (send_sync, send_sync_240, send_flags_240,
                    send_ack1, send_ack2, connected);
signal present_state : state_type := send_sync;
signal counter : std_logic_vector(23 downto 0);
signal din_sig : std_logic_vector(15 downto 0);
signal data_to_transceiver : std_logic_vector(15 downto 0);
signal link_established : std_ulogic;
signal rout_sig : std_logic_vector(15 downto 0);
signal flag_cnt : std_logic_vector(7 downto 0);
signal fifo_out_sig : std_logic_vector(15 downto 0);
signal wr_en_sig : std_logic;
signal rd_en_sig : std_logic;
signal empty_sig : std_logic;
signal rd_count_sig : std_logic_vector(6 downto 0);
signal remote_fifo_too_full : std_logic;
```

```
signal local_fifo_too_full : std_logic;
signal producer_data_in_sig : std_logic_vector(data_width - 1 downto 0);
signal consumer_data_out_sig : std_ulogic_vector(data_width - 1 downto 0);

begin

  async_fifo0: async_fifo
    port map( din => rout_sig,
              wr_en => wr_en_sig,
              wr_clk => rclk,
              rd_en => rd_en_sig,
              rd_clk => clk,
              ainit => '0',--reset_sig,
              dout => fifo_out_sig,
              full => open,
              empty => empty_sig,
              rd_count => rd_count_sig);

--state machine process to perform handshaking and establish a serial connection
process(clk)
begin
    if rising_edge(clk) then
        --allow producer to write if remote fifo is not too
        --full and link is established
        producer_os_rdy <= (not remote_fifo_too_full) and link_established;
                           --rdy signal must be synchronous to clk

        case present_state is
        when send_sync =>     --send sync pattern until local transceiver
                              --has been locked for a period of time
            sync_out <= '1';
            link_established <= '0';
            flag_cnt <= X"00";
            if local_lock_active_low = '0' then
                counter <= counter + 1;
                if counter > X"200000" then   --26 ms at 80 MHz (delay for
                                              --switch bounce)
                    counter <= X"000000";
                    present_state <= send_sync_240;
                else
                    present_state <= send_sync;
                end if;
            else
                present_state <= send_sync;
                counter <= X"000000";
            end if;

        when send_sync_240 => --send sync pattern until a valid flag
                              --pattern is received or until timeout occurs
            sync_out <= '1';
            link_established <= '0';
            if local_lock_active_low = '0' then
                if counter > X"000FFF" then
                    counter <= X"000000";
                    present_state <= send_flags_240;
                else
                    counter <= counter + 1;
                    if (rout = X"00BB") or (rout = X"00CC")
                           or (rout = X"00DD") then
                        --remote transceiver is in send_xxxx_160 or send_ackx
                        flag_cnt <= flag_cnt + 1;
                        if flag_cnt > X"F0" then
                            counter <= X"000000";
                            flag_cnt <= X"00";
                            present_state <= send_ack1;
                        end if;
                    else
                        flag_cnt <= X"00";
                        present_state <= send_sync_240;
                    end if;
                end if;
            else
                present_state <= send_sync;
            end if;

        when send_flags_240 =>  --send flag pattern until a valid flag pattern
```

```
                        --is received or until timeout occurs
    sync_out <= '0';
    din_sig <= X"00BB";
    link_established <= '0';
    if local_lock_active_low = '0' then
        if counter > X"000FFF" then
            counter <= X"000000";
            present_state <= send_sync_240;
        else
            counter <= counter + 1;
            if (rout = X"00BB") or (rout = X"00CC")
                    or (rout = X"00DD") then  --remote transceiver is in
                                              --send_xxxx_240 or send_ackx
                flag_cnt <= flag_cnt + 1;
                if flag_cnt > X"F0" then
                    counter <= X"000000";
                    flag_cnt <= X"00";
                    present_state <= send_ack1;
                end if;
            else
                flag_cnt <= X"00";
                present_state <= send_flags_240;
            end if;
        end if;
    else
        present_state <= send_sync;
    end if;

when send_ack1 =>   --wait here for remote transceiver to reach this state
                    --or the next state
    sync_out <= '0';
    link_established <= '0';
    din_sig <= X"00CC";
    if local_lock_active_low = '0' then
        if (rout_sig = X"00CC") or (rout_sig = X"00DD") then
                --remote transceiver is in send_ack1 or send_ack2
            flag_cnt <= flag_cnt + 1;
            if flag_cnt > X"F0" then
                counter <= X"000000";
                present_state <= send_ack2;
            end if;
        else
            flag_cnt <= X"00";
            present_state <= send_ack1;
        end if;
    else
        present_state <= send_sync;
    end if;

when send_ack2 => --when this state is reached both transceivers are
                  --locked, send flags for a period then go to connected state
    sync_out <= '0';
    link_established <= '0';
    din_sig <= X"00DD";
    counter <= counter + 1;
    if local_lock_active_low = '0' then
        if counter > X"0000FFF" then
            present_state <= connected;
        else
            if rout_sig = X"00DD" then
                flag_cnt <= flag_cnt + 1;
                if flag_cnt > X"F0" or counter > X"0000FFF" then
                    present_state <= connected;
                end if;
            else
                flag_cnt <= X"00";
                present_state <= send_ack2;
            end if;
        end if;
    else
        present_state <= send_sync;
    end if;

when connected =>   --both transceivers are connected, assert
                    --link_established signal and allow data transmission
    sync_out <= '0';
```

```
            din_sig <= data_to_transceiver;
            link_established <= '1';
            if local_lock_active_low = '0' then
                present_state <= connected;
            else
                counter <= X"000000";
                present_state <= send_sync;
               end if;

        when others =>
            present_state <= send_sync;

        end case;
end if;
end process;

--retrieve remote fifo too full signal off of transceiver output when link is established
with link_established select
    remote_fifo_too_full <= rout_sig(15) when '1',
                            '1' when others;

--retrieve end of stream signal off of transceiver output when link is established, consumer
enables read, and data is valid
with std_logic_vector'(link_established, consumer_rd_en, fifo_out_sig(14)) select
    consumer_eos <= fifo_out_sig(13) when "111",
                    '0' when others;

--retrieve data from fifo and output to the consumer
consumer_data_out <= consumer_data_out_sig;
with std_logic_vector'(link_established, consumer_rd_en, fifo_out_sig(14)) select
    consumer_data_out_sig <= std_ulogic_vector(fifo_out_sig(data_width - 1 downto 0))
                             when "111",
                             consumer_data_out_sig when others;

--if there is data in the fifo, allow consumer to read it
consumer_os_rdy <= not empty_sig;

--send flags and data to the transceiver for sending
data_to_transceiver(15 downto 13) <= local_fifo_too_full & producer_wr_en & producer_eos;
data_to_transceiver(data_width - 1 downto 0) <= producer_data_in_sig;

--latch data from producer when producer write enable is asserted
with producer_wr_en select
    producer_data_in_sig <= std_logic_vector(producer_data_in) when '1',
                            producer_data_in_sig when others;

--assert fifo write enable when link is established and write flag is set
with std_logic_vector'(link_established, rout_sig(14)) select
    wr_en_sig <= '1' when "11",
                 '0' when others;

--assert fifo read enable when consumer asserts read enable or when fifo data
--is not valid
rd_en_sig <= consumer_rd_en or not fifo_out_sig(14);


--latch data from DS92LV16 on rising edge of rclk when locked onto serial data stream
process(rclk)
begin
if rising_edge(rclk) then
    if local_lock_active_low = '0' then
        rout_sig <= rout;
    end if;
end if;
end process;

--update transceiver input data on falling edge of the clock
-- hardware transceiver reads input on rising clock edge so data must be
-- stable during rising edge
process(clk)
begin
if falling_edge(clk) then
    din <= din_sig;
end if;
end process;
```

```
--set local_fifo_too_full_sig based on rd_count_sig
-- flag is set if fifo contains more than 64 items and cleared if fifo contains fewer than 31
items
process(clk)
begin
if rising_edge(clk) then
    if rd_count_sig > "1000000" and rd_count_sig < "1111111" then --fifo occasionally
                                                                  --reports a count of
                                                                  --127 when it is empty
        local_fifo_too_full <= '1';
    elsif rd_count_sig < "0011111" then
        local_fifo_too_full <= '0';
    else
        local_fifo_too_full <= local_fifo_too_full;
    end if;
end if;
end process;
end Behavioral;
```