4Links SpaceWire CoDec IP Core: Product User Guide (Xilinx)

Contents

Table of Figures

Overview

This document serves as a product user guide for the 4Links SpaceWire (SpW) CoDec IP Core. In this document you will find information to help you integrate the 4Links SpW CoDec into your designs. A walkthrough for the IP Example design is included at the end of the document. The example design is used as a demonstration platform for the core. The example design can be modified as required. A powerful set of simulation procedures are included in the example design package ***SpW_Sim_lib.vhd***. These can be used to debug and verify your SpaceWire designs before implementation.

The target HDL for the SpW CoDec and testbench environment is VHDL.

Core Architecture

The 4Links SpW CoDec IP is comprised of several VHDL entities. The architecture can be split into three distinct sections; Transmit, Receive and Control. A top-level diagram of the core is included in Figure 1. For instantiating the IP,

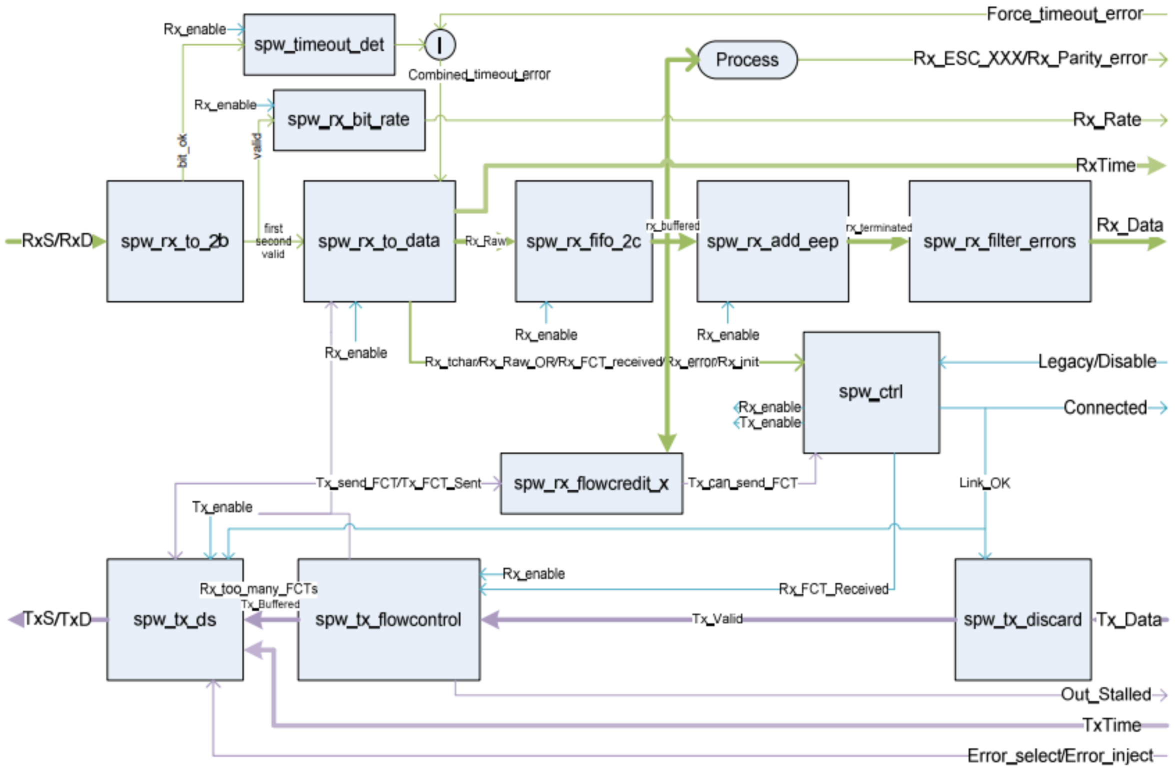


Figure 1: Top-Level Core Architecture (spw.vhd).

Core Configuration

The top-level design ***spw_wrap.vhd*** is configured using VHDL generics. There are three generics to be used.

NAME	TYPE	RANGE(LOW)	RANGE (HIGH)	UNITS
g_clock_frequency	REAL	2_000_000	N/A	Hz
g_rx_fifo_size	INTEGER	9	56	N/A
g_tx_fifo_size	INTEGER	9	56	N/A
g_mode	STRING	--	--	--

1. **g_clock_frequency**
   The clock frequency of the clock input to the IP Core (Fixed value) in Hz. For example, if using a 125MHz input clock, the value would be *125_000_000.0.* As this value is of type REAL, it must contain a decimal point to be valid. Otherwise, your toolchain may throw errors.
2. **g_rx_fifo_size**
   Depth of the RX FIFO used to store received SpW data. Recommended value is 16
3. **g_tx_fifo_size**
   Depth of TX FIFO used to send SpW data. Recommended value is 16.
4. **g_mode**
   core IO mode, choice between “diff”, “single” and “custom”. For differential, single-ended and custom IO signalling parameters.

I/O Ports

This section covers the numerous I/O ports on the 4Links SpW CoDec. Details on the function of each I/O can be found in the relevant section below.

Clock & Reset Ports

NAME	TYPE	STATUS	BRIEF
clock	std_logic	rising_edge	Positive clock source
clock_b	std_logic	rising_edge	Negative clock source (clock θ * -180°)
reset	std_logic	active high (‘1’)	IP reset input (asynchronous)

Data Ports

NAME	TYPE	STATUS	BRIEF
TX_data	std_logic_vector (8 downto 0)	Output data	Output data from SpW
Tx_OR	boolean	active high (‘1’)	Assert to load data
Tx_IR	boolean	active high (‘1’)	Asserted when ready for Tx data
RX_data	std_logic_vector (8 downto 0)	Input data	Input data to SpW
Rx_OR	boolean	active high (‘1’)	Asserted when valid Rx data
Rx_IR	boolean	active high (‘1’)	Assert to read Rx data

Rx Info Ports

NAME	TYPE	STATUS	BRIEF
RX_ESC_ESC	boolean	active high (true)	ESC-ESC Detected
RX_ESC_EOP	boolean	active high (true)	ESC-EOP Detected
RX_ESC_EEP	boolean	active high (true)	ESC-EEP Detected
RX_Parity_error	boolean	active high (true)	Parity Error Detected
RX_bits	integer (0 to 2)	N/A	Bits read on last clock
RX_rate	std_logic_vector (15 downto 0)	Output Info	Rx Channel bitrate

Time Code Ports

NAME	TYPE	STATUS	BRIEF
Rx_Time	std_logic_vector(7 downto 0)	Output Timecode	Received TC data
Rx_Time_OR	boolean	active high (true)	High when received TC data
Rx_Time_IR	boolean	active high (true)	Assert to read TC data
Tx_Time	std_logic_vector(7 downto 0)	Input Timecode	TC data to send
Tx_Time_OR	boolean	active high (true)	Assert to load TC data
Tx_Time_IR	boolean	active high (true)	High when ready for TC data

Control Ports

NAME	TYPE	STATUS	BRIEF
Disable	boolean	active high (true)	Assert to disable SpW channel
Legacy	boolean	active high (true)	Assert to use legacy spec.
Connected	boolean	active high (true)	High when SpW connected
Error_select	std_logic_vector(3 downto 0)	Select error	Select error to inject on Rx
Error_inject	boolean	active high (true)	Inject error code on SpW Rx
Out_Stalled	boolean	active high (true)	High when SpW has stalled

SpW Bypass IO Ports (custom mode only)

NAME	TYPE	STATUS	BRIEF
DDR_din_r	std_logic	Rx Data (clock)	Sampled on rising edge of clock
DDR_din_f	std_logic	Rx Data (clock_b)	Sampled on rising edge of clock_b
DDR_sin_r	std_logic	Rx Strobe (clock)	Sampled on rising edge of clock
DDR_sin_f	std_logic	Rx Strobe (clock_b)	Sampled on rising edge of clock_b
SDR_Dout	std_logic	Tx Data	SpW Tx Data Output (SDR)
SDR_Sout	std_logic	Tx Strobe	SpW Tx Strobe Output (SDR)

SpW IO Ports (diff & single modes only)

NAME	TYPE	STATUS	BRIEF
Din_p	std_logic	LVDS Din Positive	Used as Single Ended Input when “single”
Din_n	std_logic	LVDS Din Negative	
Sin_p	std_logic	LVDS Sin Positive	Used as Single Ended Input when “single”
Sin_n	std_logic	LVDS Sin Negative	
Dout_p	std_logic	LVDS Dout Positive	Used as Single Ended output when “single”
Dout_n	std_logic	LVDS Dout Negative	
Sout_p	std_logic	LVDS Sout Positive	Used as Single Ended output when “single”
Sout_n	std_logic	LVDS Sout Negative	

Using the Core

Instantiating the Core

Selecting IO Mode

The IP core has three different IO modes, “diff”, “single” and “custom”. These are configured through the ***g_mode*** generic parameter.

- Differential (LVDS) Operation “diff”.

This is the default operating mode of the core. This mode requires that the “spw_wrap_top_level” entity be edited to include device primitives for DDR registers and LVDS input buffers.

- Single Ended Operation “single”.

Single ended mode removes the need to use differential IO. All IO transactions on Data & Strobe IO will use the positive (_p) IO only. The wrapper should be modified to include device primitives for DDR registers on D/S Inputs. Outputs are SDR. This is useful if using IO-limited devices with no support for LVDS IO buffers.

- Custom IO Operation “custom”.

Custom mode bypasses the normal Din/Sin and Dout/Sout IO. Instead, this exposes the cores internal DDR register inputs and SDR outputs. These can then be connected directly to DDR registers, without the need to modify the “spw_wrap_top_level” entity. This mode is useful when designing with block GUIs or creating one-off designs on new target devices.

For small designs, the “custom” option me be preferred. However, modifying the core wrapper for your target device is encouraged. This will greatly streamline future developments with the 4Links SpaceWire IP Core. Note that for verification, the differential and single-ended modes are modelled using processes in the “spw_wrap_top_level”.

Whilst these processes accurately describe the behaviour of DDR registers and differential IO buffers, there is no guarantee that your toolchain will be able to match these processes to your target device resources. It is for this reason we insist that device primitive instantiation is used, as outlined above.

For an example of DDR registers/LVDS buffers primitive instantiation, for a Xilinx Kintex US+ FPGA, please see “spw_wrap_top_level_xilinx.vhd”.

Modifying the Core

To modify the core with your own primitives, see the relevant “generate” section of the core. For the default top-level wrapper file, these sections are:

- “diff”: lines 267 to 317.
- “single”: lines 318 to 359.
- “custom” 362 to 371.

Use the “spw_wrap_top_level_xilinx” entity as an example of adding primitives to the core generation sections. Those using Kintex UltraScale+ FPGAs may use the “_xilinx” wrapper file as is.

Connecting the Core

When using the IP Core, you must make sure that both Rx and Tx channels are connected to the target SpaceWire device. Connecting only one channel will cause the SpaceWire Link to timeout.

If the SpaceWire Link is lost, the core will automatically attempt re-connection to the SpaceWire target.

To establish a SpaceWire link, the IP core will first enter a synchronization state. Here NULL characters are sent at a low frequency (~2Mb). The end of this synchronization period is marked by the transmission of successive FCTs. Once synchronization has been performed, the IP core will begin transmitting data at the full specified rate, as defined by the clock frequency generic.

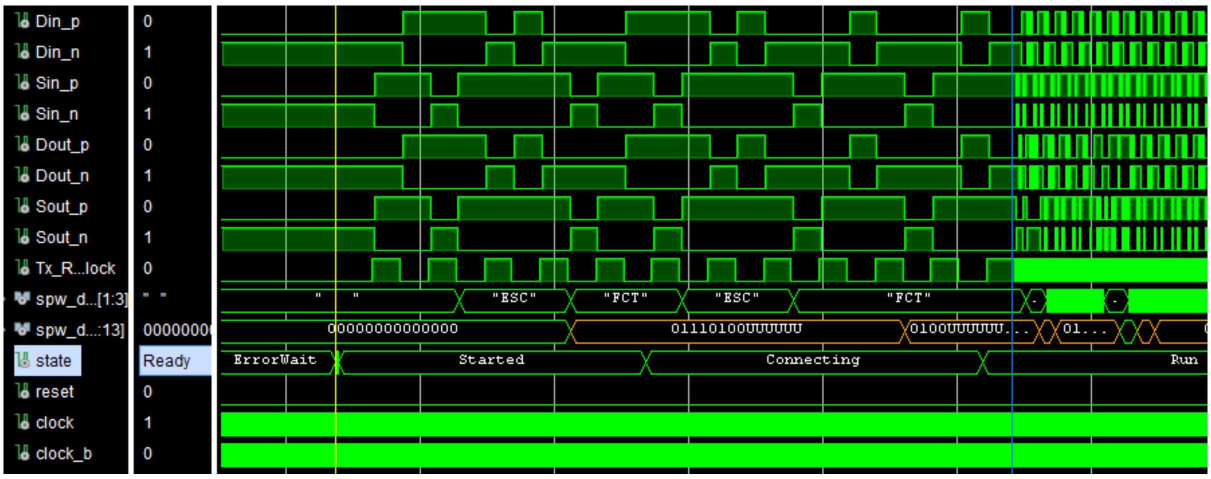


Figure 2: IP Core Start-up Synchronization

Should synchronization fail, the core will periodically attempt to re-connect to the target device until stopped by the user application.

Clock & Reset Signals

The core requires two clock signals and a single reset.

The clock signals should be generated using a PLL/MMCM/DCM (depending on required frequency and technology). ***Clock*** and ***clock_b*** should be configured so that ***clock_b*** is 180° out of phase with ***clock***. Both clocks should otherwise be identical. See Figure 3 for an example.

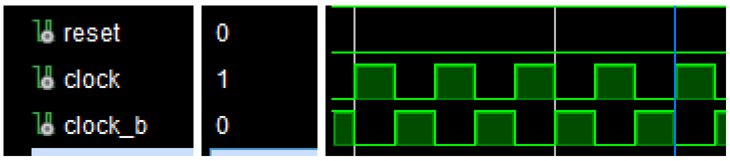


Figure 3: Core Clock(s) Example Waveform

The ***reset*** signal is asynchronous to either ***clock*** or ***clock_b***.

Note that ***clock_b*** is only used for sampling and sending data via DDR registers. All other IP functions are aligned to the rising edge of ***clock***. Therefore, when operating the core, all I/O signals should be timed with respect to the **rising edge** of ***clock***.

Core Bring-Up

To safely bring up the core, it is recommended to assert the ***reset*** input ***high*** for several clock cycles. This will ensure that any state-machine registers are in a safe state. If using a PLL/MMCM/DCM. Keep reset high for several clock cycles ***after*** the core has locked the output frequency for ***clock*** and ***clock_b***. Again, this ensures that the core starts safely. A guideline of ***4*** clock cycles is advised.

Once ready, de-assert the ***reset*** signal and wait for ***Connected*** to go ***true***. When ***connected*** is ***true*** the core is ready to send and receive data over SpW. If ***connected*** remains **false***,* check the **RxInfo** ports for debug information.

Data Channels

The Data channels use an ***Input_Ready (IR), Output_Ready (OR)*** handshake to read/write data. This is analogous to the AXI-handshake process between Master & Slave devices. A data transaction is only valid when both ***IR*** and ***OR*** ports are ***true*** on the rising edge of ***clock***. Figure 4 shows the ***IR/OR*** handshake for the ***Tx_data*** input on the core.

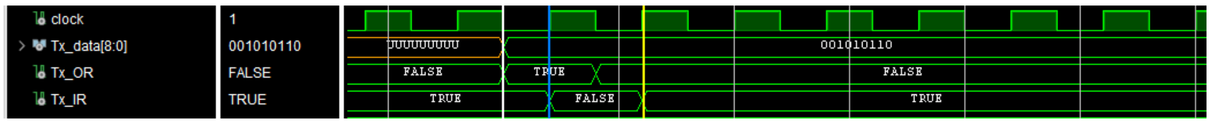


*Figure 4: Tx_data **IR/OR** Handshake Example*

Tx_IR is controlled by the IP core and ***Tx_OR*** is asserted by user logic. To load data into the Tx FiFo, connecting logic should:

- Wait for ***Tx_IR*** to be ***true***.
- Assert ***Tx_data*** and ***Tx_OR***.
- De-assert ***Tx_OR*** when ***Tx_IR*** is ***false***.

The example in Figure 4 is from a testbench, where ***Tx_OR*** changes on the falling_edge of ***clock.*** This is not required for operation; it is simply a characteristic of simulating a delta-cycle in a testbench environment. In operation, it is expected that ***Tx_OR*** will be aligned to the rising edge of ***clock***.

Figure 5 shows the waveform of the ***IR/OR*** handshake for reading ***Rx_data*** from the core.

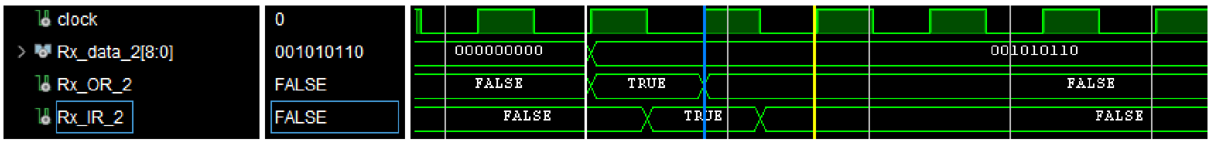


*Figure 5: Rx_data **IR/OR** handshake Example*

Note that for the cores **Rx channels**, ***OR*** is set by the core, and ***IR*** is controlled by user logic. Unlike the **Tx channels** on the core, where the opposite is true.

The data channels are 9 bits wide, where the MSB (8) is the control/data select bit and bits (7 downto 0) are the payload data bits.

Time Code Channels

The Time Code (TC) channels follow a similar pattern to the Data Channels on the core. Note that the Data channels are 9bits wide, whereas the TC channels are 8bits wide.

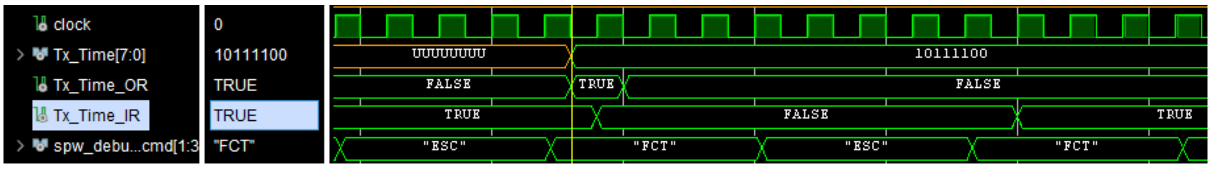


Figure 6:Tx_Time IR/OR Handshake

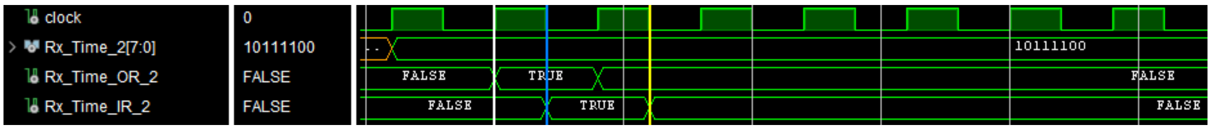


Figure 7: Rx_Time IR/OR Handshake

Sending Control Characters

Control characters, such as FCT, EOP, EEP, and ESC can be inserted on the SpW Tx channel using the ***Tx_data*** port. Here the MSB of the port (8) represents the data selection bit used in a SpW packet. When the MSB is set to '1', the data input is automatically configured to read the 2 LSB on the port. As to conform with the SpaceWire standard on control characters. Parity is automatically calculated and sent by the IP.

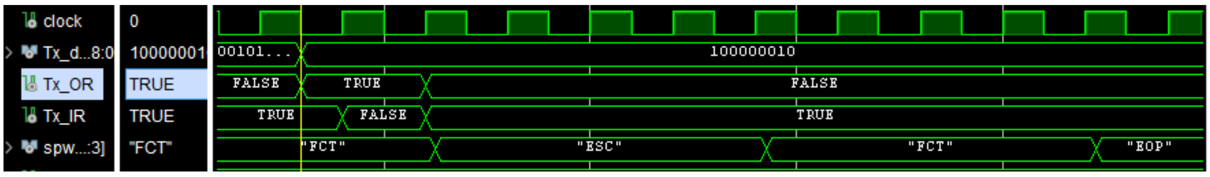


Figure 8: Send EOP Waveform Example

EEPs and EOPs will only be pushed to the IP Cores Rx_data interface if some data has been transmitted beforehand. If an EEP is sent directly after an EOP, with no data sent in between, then the core will ignore the EEP, and it will not be pushed to the Rx interface. The same is true if EEP precedes the EOP.

IP Example Design (Xilinx Vivado 2023.1)

In the IP example design, two "spw_wrap_top_level" entities are connected. One core is the "Tx" side, and the other is the "Rx" side. The signal list for the Rx side has "_2" appended to unique signals.

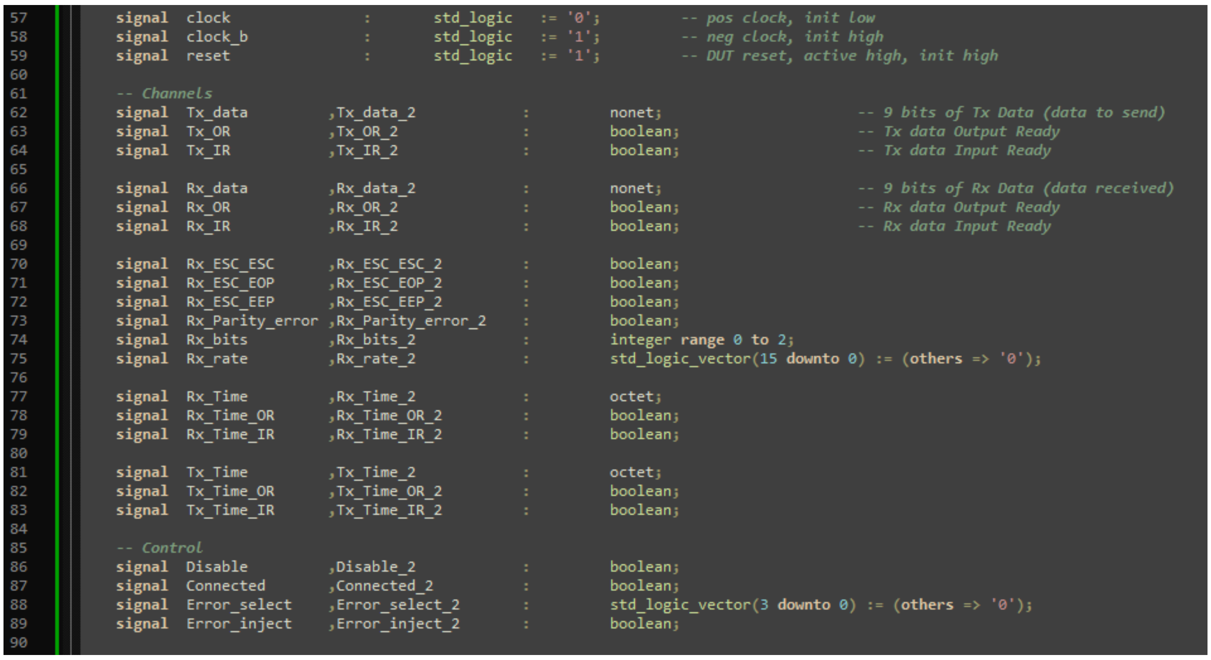
Testbench Core Configuration

The cores can be configured to operate in Single-Ended or Differential IO mode. By default, the cores will operate in Differential IO mode. To change IO modes, use the constant "c_mode" on line 54 of the testbench. Valid options are "diff", "single" and "custom", as defined in section *Selecting IO Mode.* Custom mode cannot be used without first modifying the simulation.

The single "c_mode" constant is passed to both Instantiated cores. Updating this single constant will update the IO mode of both cores automatically.

Testbench I/O & Signals

The two ***spw_wrap*** entities instantiated in the testbench are names ***SpW_DUT_tx*** and ***SpW_DUT_rx***. I/O signals from the ***SpW_DUT_rx*** entity use a ***_2*** suffix, whilst those from the ***SpW_DUT_tx*** entity have no suffix.



```vhdl
    signal  clock               :       std_logic   := '0';         -- pos clock, init low
    signal  clock_b             :       std_logic   := '1';         -- neg clock, init high
    signal  reset               :       std_logic   := '1';         -- DUT reset, active high, init high

    -- Channels
    signal  Tx_data         ,Tx_data_2          :       nonet;                      -- 9 bits of Tx Data (data to send)
    signal  Tx_OR           ,Tx_OR_2            :       boolean;                    -- Tx data Output Ready
    signal  Tx_IR           ,Tx_IR_2            :       boolean;                    -- Tx data Input Ready

    signal  Rx_data         ,Rx_data_2          :       nonet;                      -- 9 bits of Rx Data (data received)
    signal  Rx_OR           ,Rx_OR_2            :       boolean;                    -- Rx data Output Ready
    signal  Rx_IR           ,Rx_IR_2            :       boolean;                    -- Rx data Input Ready

    signal  Rx_ESC_ESC      ,Rx_ESC_ESC_2       :       boolean;
    signal  Rx_ESC_EOP      ,Rx_ESC_EOP_2       :       boolean;
    signal  Rx_ESC_EEP      ,Rx_ESC_EEP_2       :       boolean;
    signal  Rx_Parity_error ,Rx_Parity_error_2  :       boolean;
    signal  Rx_bits         ,Rx_bits_2          :       integer range 0 to 2;
    signal  Rx_rate         ,Rx_rate_2          :       std_logic_vector(15 downto 0) := (others => '0');

    signal  Rx_Time         ,Rx_Time_2          :       octet;
    signal  Rx_Time_OR      ,Rx_Time_OR_2       :       boolean;
    signal  Rx_Time_IR      ,Rx_Time_IR_2       :       boolean;

    signal  Tx_Time         ,Tx_Time_2          :       octet;
    signal  Tx_Time_OR      ,Tx_Time_OR_2       :       boolean;
    signal  Tx_Time_IR      ,Tx_Time_IR_2       :       boolean;

    -- Control
    signal  Disable         ,Disable_2          :       boolean;
    signal  Connected       ,Connected_2        :       boolean;
    signal  Error_select    ,Error_select_2     :       std_logic_vector(3 downto 0) := (others => '0');
    signal  Error_inject    ,Error_inject_2     :       boolean;
```

Figure 9: I/O Signal Declarations for IP Example Design

Ports which are shared, such as ***clock, clock_b, reset*** and the ***SpW_Tx/Rx*** channels use no suffix as only one signal declaration is required Figure 10 contains a full description of the testbench signals used in the example design.

NAME	TYPE	DESCRIPTION
Global IO Signals		
clock	std_logic	positive clock input
clock_b	std_logic	negative clock input
reset	std_logic	reset input (active high)
SpW_DUT_tx IO Signals		
Tx_data	std_logic_vector(8 downto 0)	Data to Transmit over SpW Tx Channel
Tx_OR	boolean	Output Ready Signal
Tx_IR	boolean	Input Ready Signal
Rx_data	std_logic_vector(8 downto 0)	Data Receiver over SpW Rx Channel
Rx_OR	boolean	Output Ready Signal

Rx_IR	boolean	Input Ready Signal
Rx_ESC_ESC	boolean	Asserted when ESC_ESC detected on Rx
Rx_ESC_EOP	boolean	Asserted when ESC_EOP detected on Rx
Rx_ESC_EEP	boolean	Asserted when ESC_EEP detected on Rx
Rx_Parity_error	boolean	Asserted when Parity_error detected on Rx
Rx_bits	integer 0 to 2	Number of valid SpW Rx bits on sample cycle
Rx_rate	std_logic_vector(15 downto 0)	Bitrate information for SpW Rx channel
Rx_Time	std_logic_vector(7 downto 0)	Timecode Received over SpW Rx channel
Rx_Time_OR	boolean	Output Ready Signal
Rx_Time_IR	boolean	Input Ready Signal
Tx_Time	std_logic_vector(7 downto 0)	Timecode to send over SpW Tx Channel
Tx_Time_OR	boolean	Output Ready Signal
Tx_Time_IR	boolean	Input Ready Signal
Disable	boolean	Assert to disable SpW Rx/Tx channel
Connected	boolean	Asserted when SpW UpLink is active
Error_select	boolean	Error code to select
Error_inject	std_logic_vector(3 downto 0)	Assert to inject slected Error code
SpW_DUT_rx IO signals		
Tx_data_2	std_logic_vector(8 downto 0)	Data to Transmit over SpW Tx Channel
Tx_OR_2	boolean	Output Ready Signal
Tx_IR_2	boolean	Input Ready Signal
Rx_data_2	std_logic_vector(8 downto 0)	Data Receiver over SpW Rx Channel
Rx_OR_2	boolean	Output Ready Signal
Rx_IR_2	boolean	Input Ready Signal
Rx_ESC_ESC_2	boolean	Asserted when ESC_ESC detected on Rx
Rx_ESC_EOP_2	boolean	Asserted when ESC_EOP detected on Rx
Rx_ESC_EEP_2	boolean	Asserted when ESC_EEP detected on Rx

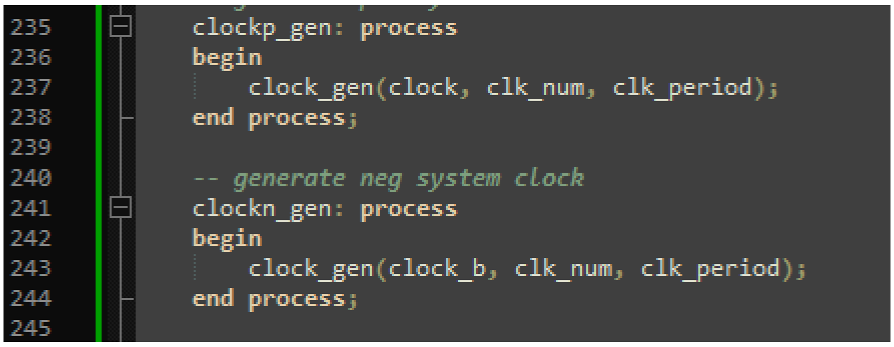
Rx_Parity_error_2	boolean	Asserted when Parity_error detected on Rx
Rx_bits_2	integer 0 to 2	Number of valid SpW Rx bits on sample cycle
Rx_rate_2	std_logic_vector(15 downto 0)	Bitrate information for SpW Rx channel
Rx_Time_2	std_logic_vector(7 downto 0)	Timecode Received over SpW Rx channel
Rx_Time_OR_2	boolean	Output Ready Signal
Rx_Time_IR_2	boolean	Input Ready Signal
Tx_Time_2	std_logic_vector(7 downto 0)	Timecode to send over SpW Tx Channel
Tx_Time_OR_2	boolean	Output Ready Signal
Tx_Time_IR_2	boolean	Input Ready Signal
Disable_2	boolean	Assert to disable SpW Rx/Tx channel
Connected_2	boolean	Asserted when SpW UpLink is active
Error_select_2	boolean	Error code to select
Error_inject_2	std_logic_vector(3 downto 0)	Assert to inject slected Error code
SpW Tx/Rx Channel LVDS Signals		
Din_p	std_logic	SpW Rx_Channel_DATA (positive)
Din_n	std_logic	SpW Rx_Channel_DATA (negative)
Sin_p	std_logic	SpW Rx_Channel_STROBE (positive)
Sin_n	std_logic	SpW Rx_Channel_STROBE (negative)
Dout_p	std_logic	SpW Tx_Channel_DATA (positive)
Dout_n	std_logic	SpW Tx_Channel_DATA (negative)
Sout_p	std_logic	SpW Tx_Channel_STROBE (positive)
Sout_n	std_logic	SpW Tx_Channel_STROBE (negative)
SpW Tx Channel Debug signals		
spw_debug_tx	std_logic_vector(8 downto 0)	SpW Tx Debug output data
spw_debug_raw	std_logic_vector(13 downto 0)	SpW Tx Debug raw output data
spw_debug_parity	std_logic	SpW Tx debug parity bit
spw_debug_cmd	string	SpW Tx debug command
spw_debug_time	std_logic_vector(7 downto 0)	SpW Tx debug timecode output

SpW Tx/Rx Channel Recovered Data Clock Signals		
Tx_Rec_Clock	std_logic	SpW Tx Recovered Data Clock
Rx_Rec_Clock	std_logic	SpW Rx Recovered Data Clock

Figure 10: SpW_ip_tb signal list

Clock Generation

The example design uses two procedures to generate ***clock*** and ***clock_b***.



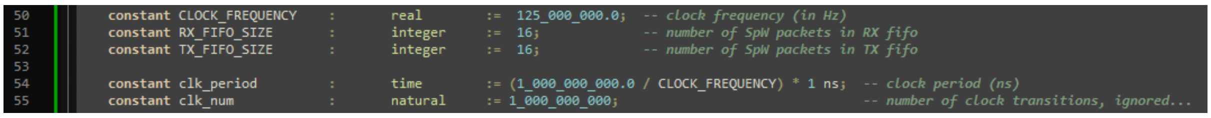
```vhdl
    clockp_gen: process
    begin
        clock_gen(clock, clk_num, clk_period);
    end process;

    -- generate neg system clock
    clockn_gen: process
    begin
        clock_gen(clock_b, clk_num, clk_period);
    end process;

```

Figure 11: Testbench Clock Generation Procedures

The procedures work by inverting the clock signal after half of the specified clock period. The clock period is calculated automatically from the ***CLOCK_FREQUENCY*** constant used in the testbench.

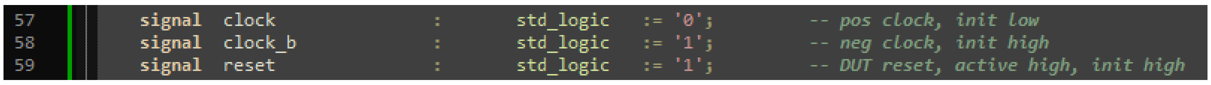


```vhdl
    constant CLOCK_FREQUENCY    :       real        :=  125_000_000.0;  -- clock frequency (in Hz)
    constant RX_FIFO_SIZE       :       integer     :=  16;             -- number of SpW packets in RX fifo
    constant TX_FIFO_SIZE       :       integer     :=  16;             -- number of SpW packets in TX fifo

    constant clk_period         :       time        := (1_000_000_000.0 / CLOCK_FREQUENCY) * 1 ns;  -- clock period (ns)
    constant clk_num            :       natural     := 1_000_000_000;                               -- number of clock transitions, ignored...
```

Figure 12: Testbench Configuration constants

For the correct clocking behaviour, the clock signals are initialized using opposite values, as seen in Figure 13.



```vhdl
    signal  clock           :       std_logic   := '0';         -- pos clock, init low
    signal  clock_b         :       std_logic   := '1';         -- neg clock, init high
    signal  reset           :       std_logic   := '1';         -- DUT reset, active high, init high
```

Figure 13: Testbench Clock initialization

Stimulus Generation

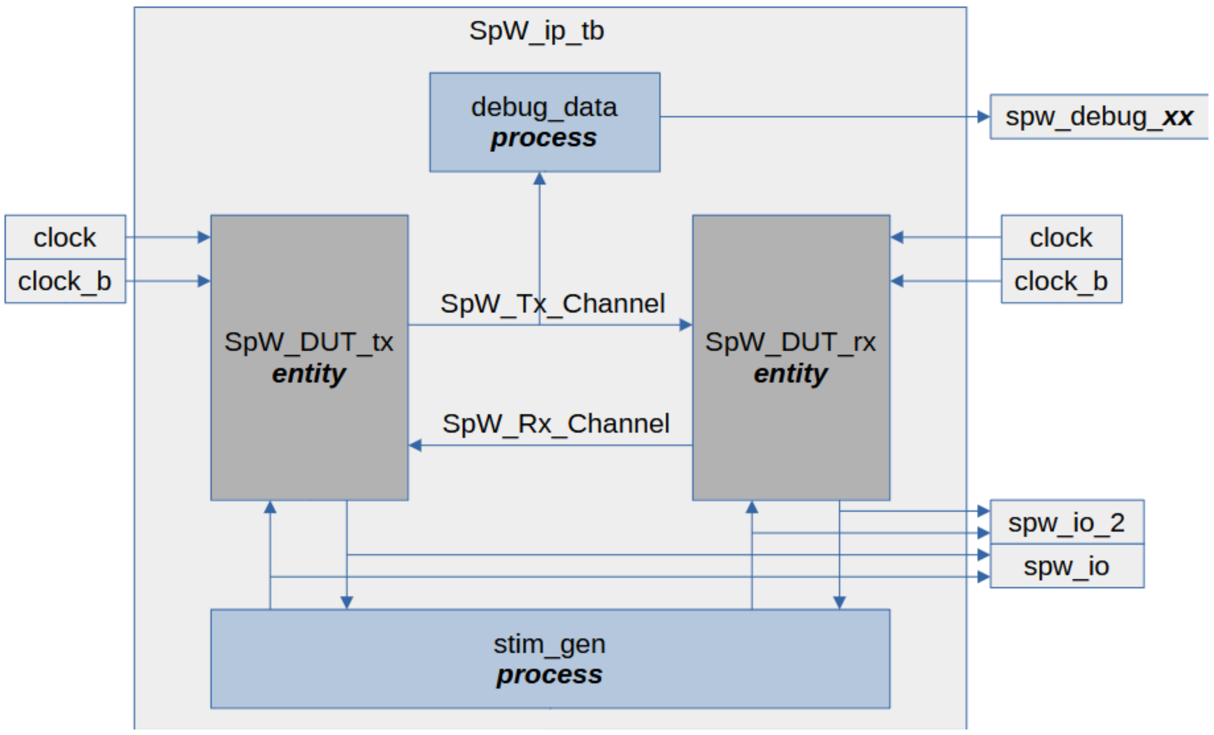


Figure 14: Testbench Basic Architecture

The ***stim_gen*** process is used to provide I/O stimulus for the two SpW CoDecs. Clock generation is accomplished using two external processes, one for each clock signal. For clarity Figure 14 does not show detailed clock connections between the IP and testbench.

The default test regime performed by ***stim_gen*** is as follows:

- Apply reset for ~67 us.
- Check valid SpW uplink was achieved
- Send SpW Tx data “001010110” from SpW_DUT_tx.
- Read SpW Rx data on SpW_DUT_rx.
- Send SpW Tx Timecode “1011100” from SpW_DUT_tx.
- Read SpW Rx Timecode on SpW_DUT_rx.
- Send SpW EOP character from SpW_DUT_tx.
- Read SpW EOP character on SpW_DUT_rx.
- Send SpW Tx data “001011110” from SpW_DUT_tx.
- Read SpW Rx data from SpW_DUT_rx.
- Send SpW EEP from SpW_DUT_tx.
- Read SpW EEP from SpW_DUT_rx.
- Send 8 Bytes of SpW Tx data from SpW_DUT_tx.
- Read 8 bytes of SpW Rx data from SpW_DUT_rx.
- Check the 8 bytes received and report errors.
- Assert Disconnect on Rx Core.
- De-assert Disconnect on Rx Core, Wait for Uplink to re-establish.

Once these procedures have completed, the process will report the stimulus has finished and the current simulation time. The report uses a severity of failure to stop the simulation immediately. The TCL console can be used to monitor the simulation status.