SpaceWire Light

version 20100910

Joris van Rantwijk
< jvrantwijk @ xs4all . nl >

http://opencores.org/project,spacewire_light

Table of Contents

1. Overview....1
2. License and disclaimer....2
3. Concept....2
   3.1. Receiver....2
   3.2. Transmitter....3
4. spwstream interface....4
   4.1. Overview....4
   4.2. Operation....5
   4.3. Configuration....5
   4.4. Interface signals....6
   4.5. Timing diagrams....7
5. Code organization....8
6. Test bench....9
7. Synthesis guidelines....9
8. Performance....10
   8.1. Link bit rate....10
   8.2. Resource utilization and timing....11

1. Overview

SpaceWire Light is a VHDL core implementing a SpaceWire encoder-decoder with FIFO interface. It is synthesizable for FPGA targets.

The goal is to provide a complete, reliable, fast implementation of a SpaceWire encoder-decoder according to ECSS-E-ST-50-12[1]. The core is "light" in the sense that it does not provide additional features such as RMAP, packet routing, etc.

This core is very suitable for application in lab environments, to add a SpaceWire interface to a custom FPGA design, and to attach a SpaceWire interface to a computer.

- ➔ Designed to conform to ECSS-E-ST-50-12C
- ➔ Developed and tested on Xilinx Spartan-3 FPGA
- ➔ Simple, byte-wide FIFO interface to RX and TX buffers
- ➔ In fully synchronous implementation: RX bit rate up to half of the system clock frequency
- ➔ With separate clock domains: RX and TX bit rate up to 4 times the system clock frequency (200 Mbit on Spartan-3)
- ➔ Test-bench included for simulation

1 SpaceWire: links, nodes, routers and networks (ECSS standard).

2. License and disclaimer

SpaceWire Light is subject to copyright 2009-2010 Joris van Rantwijk.

SpaceWire Light is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

SpaceWire Light is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with the SpaceWire Light package. If not, see <http://www.gnu.org/licenses/>.

SpaceWire Light has not been tested for strict conformance to ECSS-E-ST-50-12.
SpaceWire Light is not designed for mission critical applications.

3. Concept

The SpaceWire Light core consists of several entities, including a receiver, a transmitter, a link state machine and an application interface. This section sketches the design of the receiver and the transmitter. The next section will describe the application interface in detail.

3.1. Receiver

The receiver decodes the data/strobe signals to produce a sequence of characters and control tokens. The implementation of the receiver is split into two entities. The *front-end* of the receiver detects bit transitions on the SpaceWire line. The received bits are then transferred to the main part of the receiver, which decodes bit patterns into tokens and performs parity checking.

There are two implementations of the receiver front-end. Both are based on synchronous oversampling of the incoming signals. That is, the data and strobe signals are sampled at a fixed rate, significantly faster than the link bit rate. Bit transitions are then detected by comparing the sampled signals to previous samples. (A different approach is to extract a clock from the data/strobe signals and use that as the clock for part of the circuit. SpaceWire Light does not do this.)

An advantage of synchronous sampling is that synthesis tools are typically good at analyzing synchronous logic, while they may have problems when a clock net is driven from combinatorial logic. A disadvantage of synchronous sampling is that the sample rate must be significantly faster than the incoming bit rate to provide some margin for skew and jitter (Figure 1) (see also section 6.6 of ECSS-E-ST-50-12C). Sampling at two times the incoming bit rate seems to work well in practice.

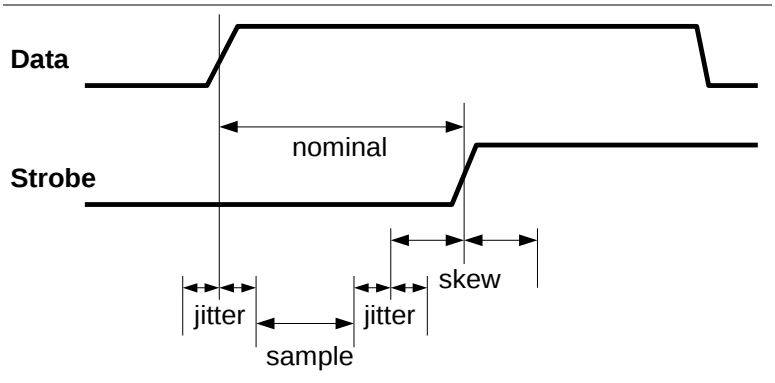

Figure 1: Effect of skew and jitter. To guarantee that all bit transitions are correctly observed, the sample period plus jitter plus skew must be less than the nominal bit period.

The generic implementation of the receiver front-end is platform independent and runs entirely in the system clock domain. It samples the incoming signals at the system clock frequency. It supports incoming bit rates up to half of the system clock frequency.

The fast implementation of the receiver front-end is designed for high bit rates (up to 4 times the system clock

frequency). It uses a separate receiver clock (*rxclk*) which may be much faster than the system clock. The incoming signals are sampled on both edges (rising and falling) of *rxclk*. Received bits are transfered to the main part of the receiver in groups of at most *rxchunk* bits per system clock cycle. This scheme supports incoming bit rates up to the *rxclk* frequency, with the additional restriction that the bit rate must be less than *rxchunk* times the system clock frequency.

3.2. Transmitter

The transmitter takes characters and control tokens and encodes them into data/strobe signals. A running SpaceWire link is never silent. When there is no useful data to transmit, the transmitter automatically sends *NULL* tokens.

Two implementations of the transmitter are available. The generic implementation runs entirely in the system clock domain. Its maximum bit rate is equal to the system clock frequency. The actual transmission rate can be set at run time to the system clock frequency divided by any integer factor[2].

The fast implementation of the transmitter is designed for high bit rates (up to 5 times the system clock frequency). It uses a separate transmit clock (*txclk*) which may be much faster than the system clock. Its maximum bit rate is equal to the txclk frequency. The actual transmission rate can be set at run time to the *txclk* frequency divided by any integer factor.

Synchronization in the fast transmitter

The fast transmitter uses two clock domains. One part is clocked by the system clock and is responsible for the interface to the rest of the system. The other part is clocked by *txclk* and implements the actual translation to bit patterns and serialization to data/strobe signals. The txclk is typically much faster than the system clock. Synchronization is needed to safely transfer data between these domains.

The system clock domain contains a buffer of 2 slots in which it puts tokens to be transmitted. The txclk domain takes tokens from the buffer, alternating between the two slots, and updates flags to tell the sysclk domain when it must refill a slot in the buffer.

Ideally, the sysclk domain should refill a used slot in the buffer *before* the txclk domain tries to take the next token from that same slot. However, it may happen that the txclk domains tries to get the next token while the slot has not yet been refilled. This is a bit of a problem for the txclk domain because it *must* start the transmission of a new token but it does not know which token. This is solved by inserting a NULL token into the outgoing stream. In fact, this is the normal way in which the fast transmitter sends NULL tokens when there truly is no data to send. On the other hand, it would be unfortunate if there were data available in the sysclk domain but for some reason the buffer slot was not refilled quickly enough. In that case the transmitter would be spontaneously inserting NULL tokens into the data flow, which is inefficient use of the link bit rate.

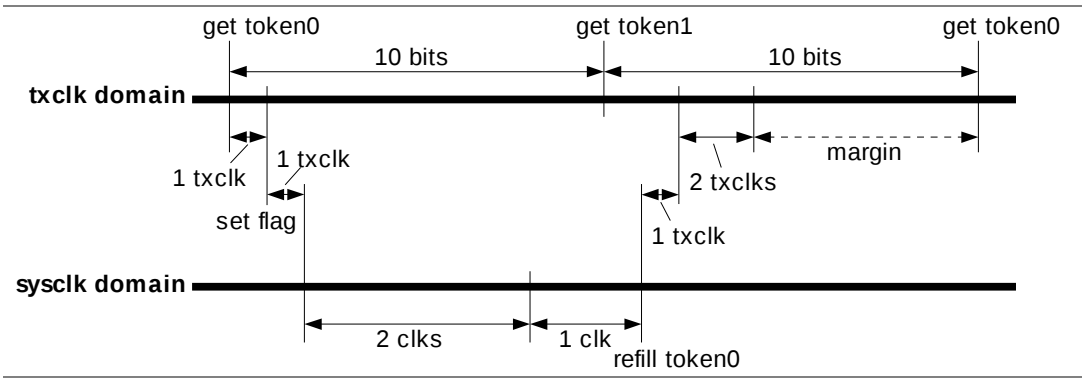

Figure 2: Synchronization in the fast transmitter

By analyzing the synchronization between the two clock domains, we can ensure that buffer slots are refilled fast enough in nearly all situations, provided that the link bit rate is at most 5 times the system clock frequency. This guarantees that the transmitter will not spontaneously insert NULL tokens into the data flow. Figure 2 shows the steps that occur between taking a token from a buffer slot and refilling that slot:

- The txclk domain sets a flag to indicate that the slot has been used (1 txclk)
- Wire delay between the clock domains (1 txclk)
- Two-stage synchronizer in the system clock domain (2 sysclk)

2 The minimum bit rate for SpaceWire is 2 Mbit/s. Link initialization is always done at 10 Mbit/s.

- The system clock domain refills the buffer slot and sets a flag to show that it has done so (1 sysclk)
- Wire delay between the clock domains (1 txclk)
- Two-stage synchronizer in the system clock domain (2 txclk)

Adding up to 5 txclk + 3 sysclk periods.

After the txclk domain takes one token from a buffer slot, it will take a token from the other slot before again taking a token from the first slot. Since normal data characters are 10 bits long. there will usually be 20 bit periods between accesses to the same slot. To guarantee a timely refill of the buffer slot, we need to show that 5 txclks + 3 sysclks ≤ 20 bits. A bit period is at least one txclk period, therefore the inequality holds provided that the bit rate is at most 5 times the system clock frequency.

The above analysis only covers transmission of normal data characters, which are 10 bits long. Since an EOP token is only 4 bits long, the transmitter may need to insert a NULL after every EOP at certain bit rates. For FCT tokens, a trick is used to avoid the issue. An FCT can be placed in a buffer slot together with a normal character. In this case the FCT does not take up a buffer slot of its own, therefore it will not cause the insertion of a NULL token.

4. spwstream interface

4.1. Overview

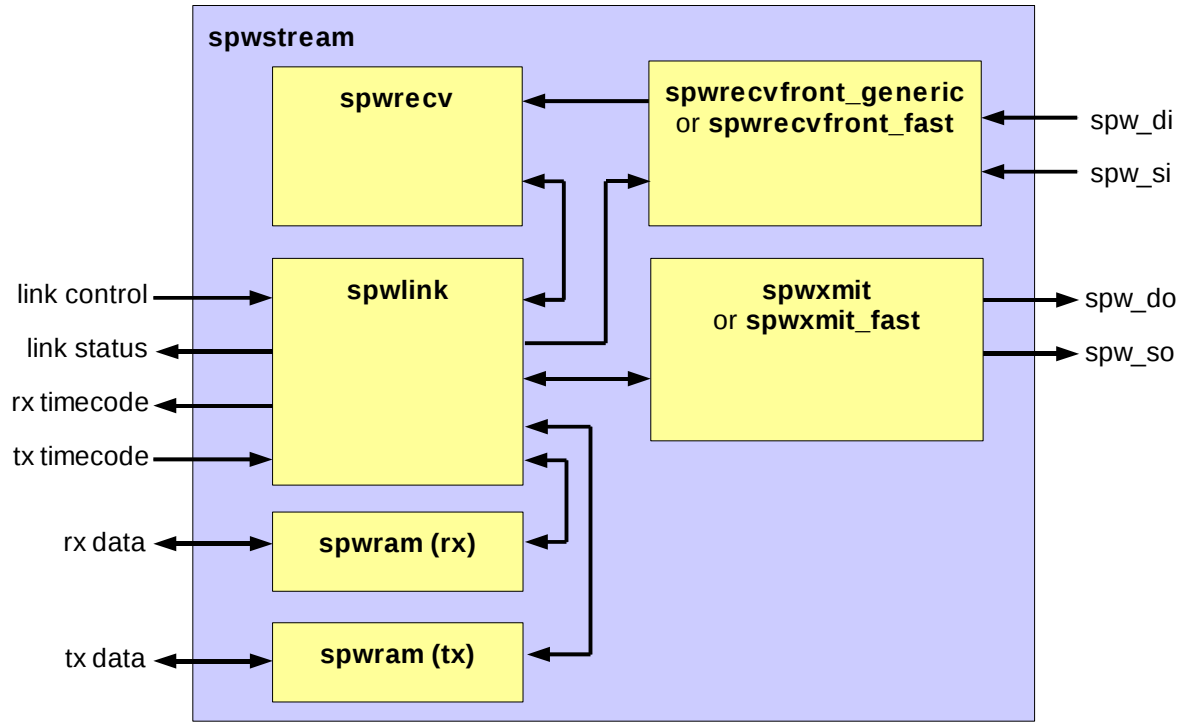

Figure 3: Block diagram of spwstream entity

The entity *spwstream* encapsulates a complete SpaceWire codec with a simple FIFO-based interface. This entity is intended to be used by applications. It consists of a receiver, transmitter, link state machine, FIFOs and some glue logic. Incoming and outgoing data streams are handled as sequences of characters. A character is either a normal data byte or an end-of-packet marker.

The FIFOs are dual-port RAM blocks, 9 bits wide (8 data bits and 1 flag bit). The depth of the FIFOs is configurable. The FIFOs retain their contents even when the link is reset. Only an explicit reset of the core will clear the FIFOs.

4.2. Operation

Link state management

The link is managed by a finite state machine as specified by the SpaceWire standard. When the control signal *linkstart* is high, the state machine attempts to establish a link with the remote interface. If an error occurs (parity error, disconnection or other error) the link is reset. A broken link will be re-established if *linkstart* is still high.

Note: It is not possible to establish a link when the receive FIFO is (almost) full. The reason is that at least one FCT token must be sent as part of the handshake procedure.

Receiving

Received characters (data bytes or EOP/EEP markers) are placed in the receive FIFO until the the application reads them out. Reading from the FIFO is managed by a simple handshake mechanism.

Flow control is handled transparently: when the receive FIFO fills up, the core will pause transmission of FCT tokens to the remote interface.

If a link error occurs in the middle of an incoming packet, a synthetic EEP character is inserted in the receive FIFO to mark the fact that the packet may be incomplete. This happens even if the link was explicitly shut down through a *linkdisable* pulse.

Transmitting

The application writes outgoing characters (data bytes or EOP/EEP markers) into the transmit FIFO. Writing to the FIFO is managed by a simple standard handshake mechanism. Characters are taken from the FIFO and transmitted on the SpaceWire link whenever the link is running and there is flow control credit available.

If a link error occurs in the middle of an outgoing packet, subsequent outgoing characters are discarded up to and including the next EOP/EEP character. If there is no EOP/EEP character in the transmit FIFO, data discarding remains in effect until the next EOP/EEP is queued for transmission or until the link is explicitly shut down. The purpose of data discarding is to avoid sending a partial packet once the link is re-established. Data discarding is not performed if the link is explicitly shut down through a *linkdisable* pulse.

The bit rate of the outgoing signal is generated by a clock divider, either from the system clock (generic implementation) or from a dedicated transmission clock (fast implementation). The clock division factor can be changed at run time. Link initialization is however always done at 10 Mbit/s.

Time-codes

Transmission of a time-code is triggered through a pulse on the *tick_in* signal. The time-code is buffered inside the core until it can be transmitted. Transmission of time-codes has priority over data characters.
A received time-code is announced through a pulse on the *tick_out* signal. Handling of received time-codes does not fully conform to ECSS-E-ST-50-12C. The *tick_out* signal as implemented by SpaceWire Light produces a pulse for every received time-code, while ECSS-E-ST-50-12C (section 8.12) specifies that a pulse should only be produced if the received time-code value is exactly one more than the previous received value. If needed, strictly conforming behaviour can be obtained by filtering received time-codes in additional logic external to the *spwstream* entity.

4.3. Configuration

The entity is configured through the following VHDL generics:

name	**description**	**type**	**default**
`sysfreq`	Frequency of the system clock in Hz. Must match the frequency of the clk signal. Used for internal timing of handshake and timeouts and to compute the clock division factor for 10 Mbit/s during the handshake phase.	real	
`txclkfreq`	Frequency of the txclk signal in Hz. Used to compute the clock division factor for 10 Mbit/s during the handshake phase. Only needed when `tx_impl` = `impl_fast`.	real	0

rximpl	Selection of a receiver front-end implementation. Select impl_generic for synchronous receiving in the system clock domain; rxclk will not be used; maximum bitrate will be half of the system clock frequency. Select impl_fast for synchronous receiving in the rxclk domain; maximum bit rate will be the either the rxclk frequency or rxchunk times the system clock frequency, whichever is lower.		impl_generic
rxchunk	In case of impl_fast, determines the maximum number of bits that can be received per system clock tick. Incoming bit rate is thus limited to rxchunk times the system clock frequency. Higher values for rxchunk put more stress on circuit timing. In case of impl_generic, rxchunk must be set to 1.	integer, 1 .. 4	1
tximpl	Selection of a transmitter implementation. Select impl_generic for transmission in the system clock domain. Select impl_fast for transmission in the txclk domain.		impl_generic
rxfifosize_bits	Size of the receive FIFO as the 2-logarithm of the number of bytes.	integer, 6 .. 14	11 (2 kByte)
txfifosize_bits	Size of the transmit FIFO as the 2-logarithm of the number of bytes.	integer, 2 .. 14	11 (2 kByte)

4.4. Interface signals

The entity uses interface signals defined by the following VHDL ports:

name	dir	description
clk	I	System clock, active on rising edge.
rxclk	I	Receiver sample clock (only when rximpl = impl_fast).
txclk	I	Transmit base clock (only when tximpl = impl_fast).
rst	I	Synchronous reset (active high).
autostart	I	Enables automatic link start on receipt of a NULL character.
linkstart	I	Enables link start once the Ready state is reached (overrides autostart).
linkdis	I	Do not start link; disconnect a running link (overrides linkstart and autostart).
txdivcnt<7:0>	I	Scaling factor minus 1, used to derive the transmit bit rate from the transmit base clock. The base clock is the system clock when tximpl = impl_generic, or txclk when tximpl = impl_fast. The base clock is divided by (unsigned(txdivcnt) + 1). During link setup, the transmission bit rate is always 10 Mbit/s regardless of this signal.
tick_in	I	Pulse high for one clock cycle to request transmission of a time-code.
ctrl_in<1:0>	I	Control bits to be sent with the time-code. Must be valid when tick_in is high.
time_in<5:0>	I	Counter value to be sent with the time-code. Must be valid when tick_in is high.
txwrite	I	Set high by the application to write a character to the transmit FIFO. A character is stored in the FIFO whenever txwrite and txrdy are both high on the rising edge of clk. This signal is ignored while txrdy is low.
txflag	I	Control flag to be sent with the next character. Set low to send a data byte, or high to send EOP or EEP. Must be valid while txwrite is high.
txdata<7:0>	I	Data byte to be sent (txflag=0), or 0x00 for EOP or 0x01 for EEP (txflag=1). Must be valid while txwrite is high.
txrdy	O	High if the entity is ready to accept a character for the transmit FIFO.
txhalff	O	High if the transmit FIFO is at least half full.
tick_out	O	High for one clock cycle if a time-code was just received.
ctrl_out<1:0>	O	Control bits of the last received time-code.
time_out<5:0>	O	Counter value of the last received time-code.
rxvalid	O	High if rxflag and rxdata contain valid data (i.e. when the receive FIFO is not empty).
rxhalff	O	High if the receive FIFO is at least half full.

rxflag	O	High if the received character is EOP or EEP; low if the received character is a data byte. Valid if rxvalid is high.
rxdata<7:0>	O	Received byte (rxflag=0) or 0x00 for EOP or 0x01 for EEP (rxflag=1). Valid if rxvalid is high.
rxread	I	Set high by the application to accept a received character. A character is removed from the receive FIFO whenever rxvalid and rxread aro both high on the rising edge of clk.
started	O	High if the link state machine is in the Started state.
connecting	O	High if the link state machine is in the Connecting state.
running	O	High if the link state machine is in the Run state, indicating that the link is operational. If started, connecting and running are all low, the link is in an initial state with the transmitter disabled.
errdisc	O	Disconnection detected in the Run state. Triggers a link reset; auto-clearing.
errpar	O	Parity error detected in the Run state. Triggers a link reset; auto-clearing.
erresc	O	Invalid escape sequence detected in the Run state. Triggers a link reset; auto-clearing.
errcred	O	Credit error detected. Triggers a link reset; auto-clearing.
spw_di	I	Data In signal from SpaceWire link to core.
spw_si	I	Strobe In signal from SpaceWire link to core.
spw_do	O	Data Out signal from core to SpaceWire link.
spw_so	O	Strobe Out signal from core to SpaceWire link.

4.5. Timing diagrams

The following diagrams illustrate the timing of the spwstream interface. All inputs and outputs, except for the SpaceWire signals, are synchronous to the rising edge of the system clock. Signals are not guaranteed to be glitch-free in the periods between active clock edges (but such glitches are irrelevant in a synchronous system that meets all timing constraints).

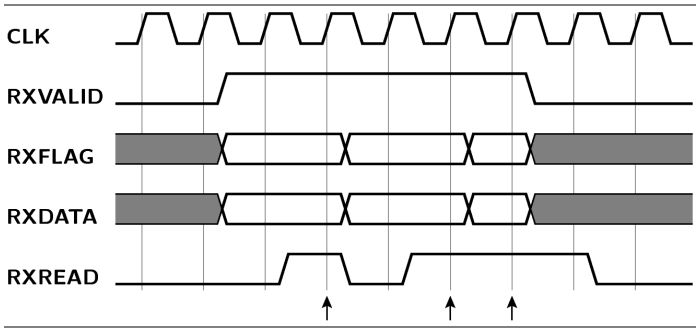

Figure 4: Timing of reading from receive FIFO

Reading from the receive FIFO is controlled by handshake signals *rxvalid* and *rxread*. A character is taken from the FIFO whenever both *rxvalid* and *rxread* are high at the rising edge of the system clock. This mechanism allows both sides to pause the transfer at any time. In figure 4, three characters are transfered at the clock cycles marked with arrows.

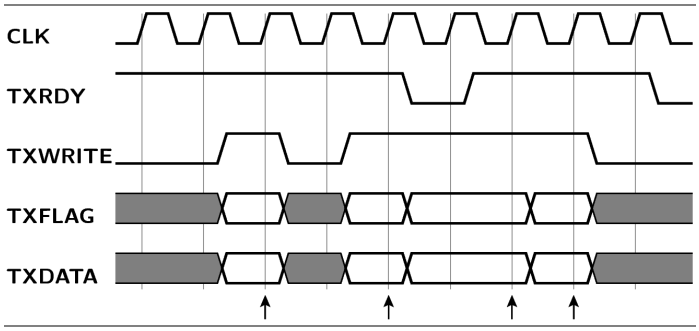

Figure 5: Timing of writing to transmit FIFO

Writing to the transmit FIFO is controlled by handshake signals *txrdy* and *txwrite*. A character is written to the FIFO whenever both *txrdy* and *txwrite* are high at the rising edge of the system clock. In figure 5, four characters are transfered at the clock cycles marked with arrows.

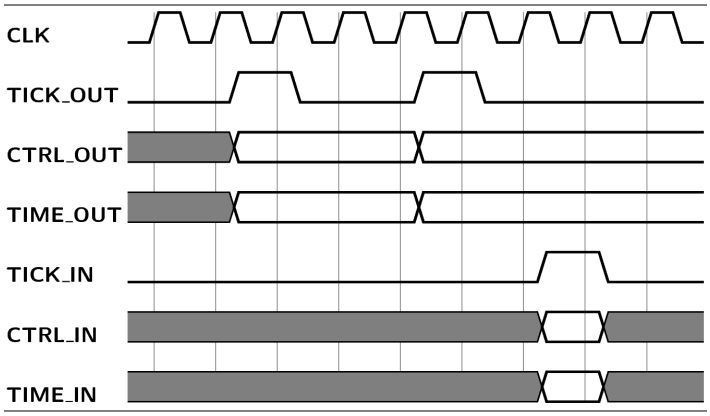

Figure 6: Timing of receiving/transmitting time-codes

A received time-code is announced by a single-cycle pulse on *tick_in*. The value of the last received time-code is available on *time_out* and *ctrl_out*. To send a time-code, the application prepares *time_in* and *ctrl_in* and gives a single-cycle pulse on *tick_in*. The outgoing time-code is buffered internally in the core until it can be transmitted. Figure 6 illustrates two incoming time-codes (impossibly close after each other) and one outgoing time-code.

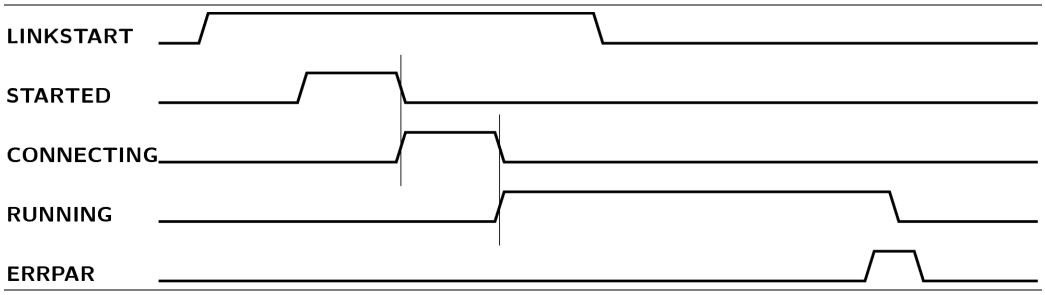

Figure 7: Sequence of link setup and status reporting (time not to scale).

5. Code organization

`rtl/vhdl/`	Synthesizable VHDL code.
`spwpkg.vhd`	Entity declarations as a VHDL package.
`spwstream.vhd`	Application interface in terms of character streams.
`spwlink.vhd`	Control logic for exchange level and link management.
`spwrecv.vhd`	Receiver / token decoder.
`spwrecvfront_generic.vhd`	Front-end for receiver, generic implementation.
`spwrecvfront_fast.vhd`	Front-end for receiver; fast implementation with separate rx clock.
`spwxmit.vhd`	Transmitter; generic implementation.
`spwxmit_fast.vhd`	Transmitter; fast implementation with separate tx clock.
`spwram.vhd`	Synchronous dual-port RAM block.
`streamtest.vhd`	Test driver for *spwstream*; not normally used in applications.
`bench/vhdl`	Test bench code.
`spwlink_tb.vhd`	Test bench for spwlink, spwrecv, spwxmit.
`spwlink_tb_all.vhd`	Several instances of *spwlink_tb* in different configurations.
`streamtest_tb.vhd`	Test bench for *spwstream*, based on the test driver *streamtest*.

`sim/ghdl`	RTL simulation with GHDL
`syn/spwstream_gr-xc3s1500`	Example design Spartan-3 on GR-XC3S1500 board.
`syn/streamtest_gr-xc3s1500`	Test design for Xilinx Spartan-3 on GR-XC3S1500 board.
`syn/streamtest_digilent-xc3s200`	Test design for Xilinx Spartan-3 on Digilent XC3S200 board.

6. Test bench

Two test benches are included. The purpose of *spwlink_tb* is to test the receiver, transmitter and link state machine at the level of SpaceWire signalling. It instantiates a codec and feeds it a simulated SpaceWire signal while monitoring the outgoing SpaceWire signal. A number of scenarios are simulated, including link setup, character transmission and flow control.

To verify the operation of the codec in various modes and bit rates, *spwlink_tb* should be simulated several times with different configuration parameters. This is exactly the purpose of *spwlink_tb_all*. It creates a number of instances of *spwlink_tb* with varying parameters and simulates them one by one.

The second test bench, *streamtest_tb* is intended to test the FIFO management in *spwstream* and the general flow of data through an operational SpaceWire link. It instantiates *streamtest*, which is a synthesizable test driver for *spwstream*. The SpaceWire side of the codec is looped back to itself, such that any transmitted data flows back as incoming data. The test driver will transmit a pattern of packets and time-codes through *spwstream*, checking that the received data stream is as expected.

Note that *streamtest* may also be used in the top-level of an FPGA design in order to test an implementation of the core on a physical board.

The test benches have been developed with GHDL[3] (version 0.29) but will presumably also work with other VHDL simulators. A Makefile is provided to run the simulations in GHDL.

For example, the following commands will run both test benches:

```
make test_spwlink
make test_streamtest
```

7. Synthesis guidelines

Platform support

The core has until now only been tested on Xilinx Spartan-3 (using Xilinx ISE 11.2).

The code should be portable to other FPGA platforms. There may be some issues, especially in the following areas:

- inference of block RAM in *spwram.vhd*;
- clock domain synchronization and Xilinx-specific attributes in *spwrecvfront_fast.vhd*;
- clock domain synchronization and Xilinx-specific attributes in *spwxmit_fast.vhd*.

Clocking

A system clock (*clk*) must be provided to the core. The system clock frequency should typically be at least 20 MHz in order to support 10 Mbit SpaceWire links as required. If separate rxclk/txclk are used, the system clock frequency may be lower.

If the fast receiver front-end is used, a separate receive clock (*rxclk*) is needed. If the fast transmitter is used, a separate transmit clock (*txclk*) is needed. Both *rxclk* and *txclk* should be at least as fast as the system clock and may be much faster than the system clock. The receive and/or transmit clock may be equal to the system clock, or derived from the system clock using a PLL or DCM, or derived from a free running oscillator.

Clock frequencies should be such that the SpaceWire core is able to send and receive at 10 Mbit/s (±10%) to correctly perform link initialization. See §3 for the relation between clock frequency and bit rate.

Timing constraints

Timing constraints will be needed to ensure correct timing of the synthesized circuit. Period constraints are needed on all clock nets. If the fast implementation of the receiver and/or transmitter are used, the core will consist of multiple

3 The GNU VHDL simulator, http://ghdl.free.fr/

clock domains. In this case, path constraints are needed on all paths that cross between clock domains. The cross-domain paths must be constrained to the period of the faster of the two clocks.

For example, the following constraints could be used with Xilinx tools for a 40 MHz system clock, 100 MHz receive clock and 125 MHz transmit clock. See also the `.UCF` files used by the example designs.

```
NET "clk" TNM_NET = "clk";
NET "rxclk" TNM_NET = "rxclk";
NET "txclk" TNM_NET = "txclk";
TIMESPEC "TS_clk" = PERIOD "clk" 25 ns;
TIMESPEC "TS_rxclk" = PERIOD "rxclk" 10 ns;
TIMESPEC "TS_txclk" = PERIOD "txclk" 8 ns;
TIMESPEC "TS_rx_to_sys" = FROM "rxclk" TO "clk" 10 ns;
TIMESPEC "TS_sys_to_rx" = FROM "clk" TO "rxclk" 10 ns;
TIMESPEC "TS_tx_to_sys" = FROM "txclk" TO "clk" 8 ns;
TIMESPEC "TS_sys_to_tx" = FROM "clk" TO "txclk" 8 ns;
```

Synthesis log files should be reviewed to check that the synthesized circuit meets all timing constraints. It is quite possible that some constraints fail if very fast clocks are used (e.g. 200 MHz on Spartan-3).

I/O registers

To minimize skew between the data and strobe lines, it is important that the *Data_In* and *Strobe_In* are sampled at precisely the same time. Any difference in wire delay from the input pads to the input flip-flops adds to the total data/strobe skew. Similarly, Data_Out and Strobe_Out should be updated at precisely the same time. Any difference in wire delay from the output flip-flops to the output pads adds to the total data/strobe skew.

Certain FPGA families (including Xilinx Spartan) have an option to implement input and output flips-flops inside the I/O element. This option should be enabled, since it makes the input/output delays much more predictable. Alternatively, timing constraints could be used to limit input/output delays, but this is harder and less effective.

To enable I/O registers for the Xilinx command line tools, specify "`-pr b`" as an option to the mapper.
To enable I/O registers in Xilinx ISE, got to Implement, Mapper, Process Options, and set "Pack I/O registers into IOBs" to "Inputs and Outputs".

8. Performance

8.1. Link bit rate

The maximum link bit rate supported by the core depends on its configuration and on the clock frequencies used. The relation between clock frequency and link bit rate is discussed in §3.

Max RX bit rate	when rximpl = impl_generic:	~ half system clock frequency
	when rximpl = impl_fast:	**min**(~ *rxclk* frequency , *rxchunk* times system clock frequency)
Max TX bit rate	when tximpl = impl_generic:	system clock frequency
	when tximpl = impl_fast:	*txclk* frequency

The following results were obtained with *spwstream*, implemented on a Xilinx XC3S2000-4. The core was tested by setting up a SpaceWire link between the test board and a commercial SpaceWire interface product.

Configuration	rximpl = impl_fast; tximpl = impl_fast; rxchunk = 4
Clock frequency	system clock = 60 MHz; rxclk = 200 MHz; txclk = 200 MHz
Maximum TX bit rate	200 Mbit/s
Maximum RX bit rate	tested up to 200 Mbit/s
Payload data rate	18 MByte/s (at 200 Mbit/s link rate)

8.2. Resource utilization and timing

The number of gates needed to implement the core, as well as the maximum clock frequency supported by the core, depend on its configuration and on the FPGA platform.

The following results were obtained with *spwstream* as the top-level entity on a Xilinx XC3S1500-4. The core was synthesized with Xilinx ISE 11.3.

Configuration	rximpl = impl_generic tximpl = impl_generic	rximpl = impl_fast tximpl = impl_fast rxchunk = 4
Clock frequency	system clock = 100 MHz	system clock = 75 MHz rxclk = 240 MHz txclk = 240 MHz
Nr of flip-flops	163	366
Nr of LUTs	426	782
Nr of slices	245	487
Nr of block RAMs	2	2

Note that the FPGA was in this case nearly empty. It will be more difficult for the synthesizer to meet timing goals if the utilization ration of the FPGA is high. The clock frequencies shown above may therefore not be feasible if the core is attached to a complex digital design.