SpaceWire Light

version 20130504

Joris van Rantwijk
`< joris @ jorisvr . nl >`

http://opencores.org/project,spacewire_light

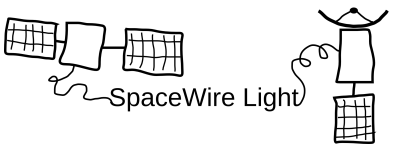

Table of Contents

1. Overview....2
2. License and disclaimer....2
3. Concept....3
   3.1. Receiver....3
   3.2. Transmitter....3
4. spwstream interface....5
   4.1. Overview....5
   4.2. Operation....5
   4.3. Configuration....6
   4.4. Interface signals....7
   4.5. Timing diagrams....8
5. AMBA interface....10
   5.1. Overview....10
   5.2. Operation....10
   5.3. DMA interface....11
   5.4. Registers....14
   5.5. Configuration....15
   5.6. Interface signals....16
6. Code organization....17
7. Test bench....18
8. Synthesis guidelines....19
   8.1. Platform support....19
   8.2. Clocking....19
   8.3. Timing constraints....19
   8.4. I/O registers....19
   8.5. Integration in GRLIB....20
9. Software....21
   9.1. Stand-alone test program....21
   9.2. RTEMS driver....21
10. Performance....22
   10.1. Link bit rate....22
   10.2. Resource utilization and timing of spwstream interface....22
   10.3. Resource utilization of spwamba interface....23

1. Overview

SpaceWire Light is a VHDL core implementing a SpaceWire encoder-decoder, synthesizable for FPGA targets.

This project is aimed at lab environments, SpaceWire interfaces for custom FPGA designs and SpaceWire interfaces for computers. 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.

- ➔ Designed to conform to ECSS-E-ST-50-12C
- ➔ Developed and tested on Xilinx Spartan-3 and Virtex-5 FPGAs
- ➔ Simple, byte-wide FIFO interface to RX and TX buffers
- ➔ Optional AMBA bus interface, compatible with LEON3/GRLIB[2]
- ➔ 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

2. License and disclaimer

SpaceWire Light is subject to copyright 2009-2011 Joris van Rantwijk.

SpaceWire Light is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 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 General Public License for more details.

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

In addition, the parts of SpaceWire Light which do not depend on GRLIB may be distributed 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. To clarify: the *spwstream* interface, and all files that are necessary to implement it, may be distributed under either GPL or LGPL; the *spwamba* interface, which depends on GRLIB, may only be distributed under GPL.

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.

1 SpaceWire: links, nodes, routers and networks (ECSS standard).
2 GRLIB is a library of cores for system-on-chip development, designed by Aeroflex Gaisler.

3. Concept

The SpaceWire Light core consists of several entities, including a receiver, a transmitter, a link state machine and two application interfaces. This section sketches the design of the receiver and the transmitter. Following sections will describe the application interfaces 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.

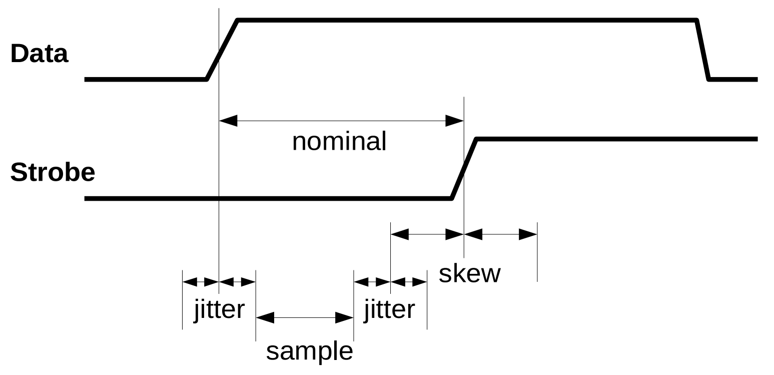

Figure 1: Effect of skew and jitter. To guarantee that all bit transitions are correctly observed, the sample period plus jitter plus skew must be less than the nominal bit period.

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[3].

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

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.

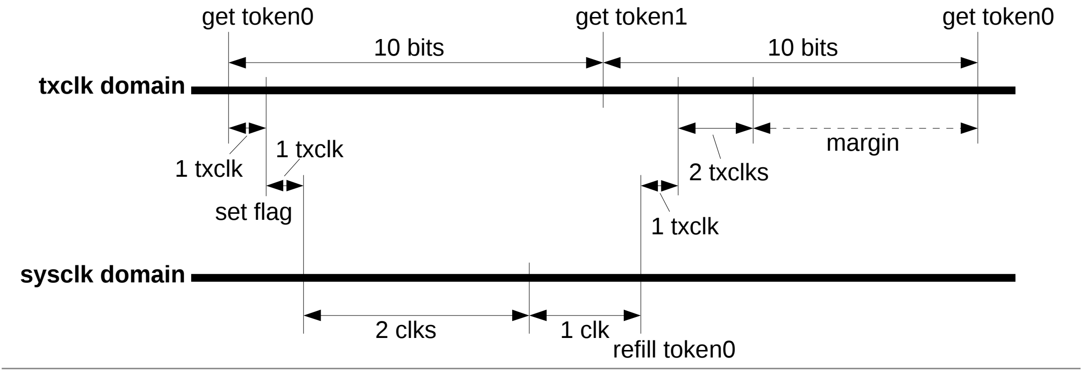

Figure 2: Synchronization in the fast transmitter

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)
- 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

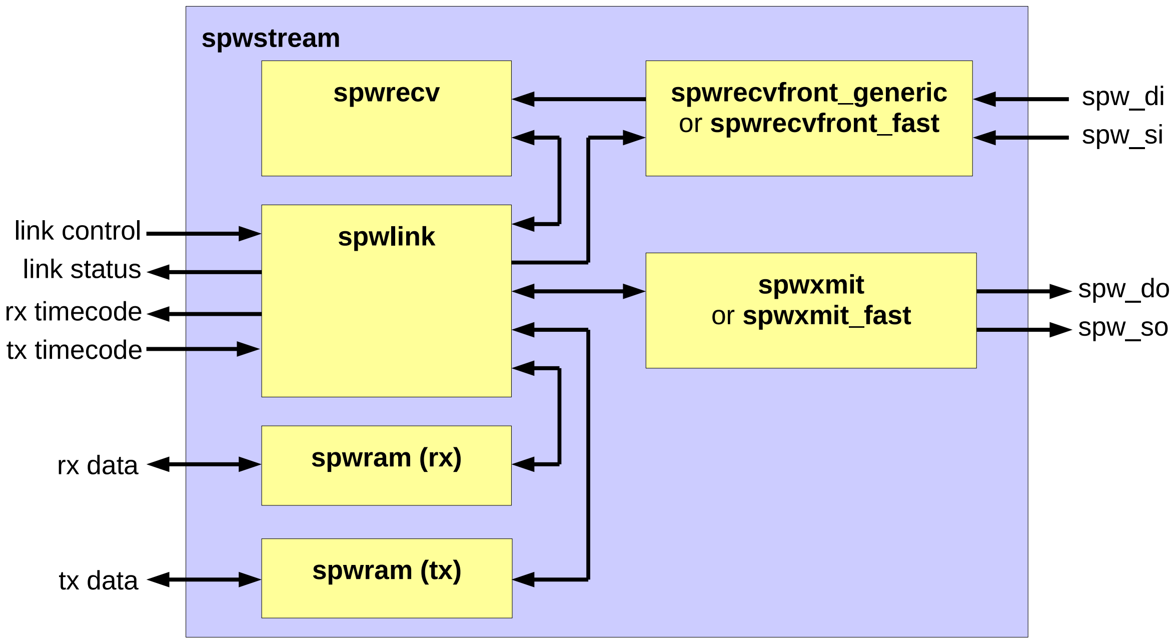

Figure 3: Block diagram of spwstream entity

The entity *spwstream* encapsulates a complete SpaceWire codec with a simple FIFO-based interface. This entity is one of the two application interfaces provided by SpaceWire Light. 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 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 handshake mechanism. Characters are taken from the FIFO and transmitted on the SpaceWire link, provided that 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 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).

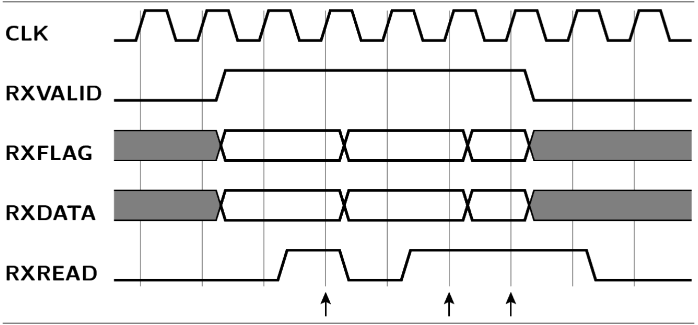

Figure 4: Timing of reading from receive FIFO

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.

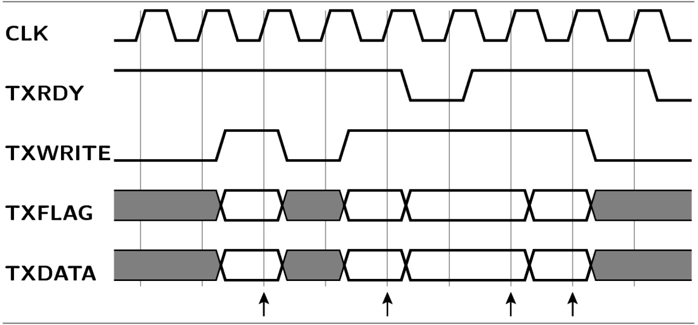

Figure 5: Timing of writing to transmit FIFO

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.

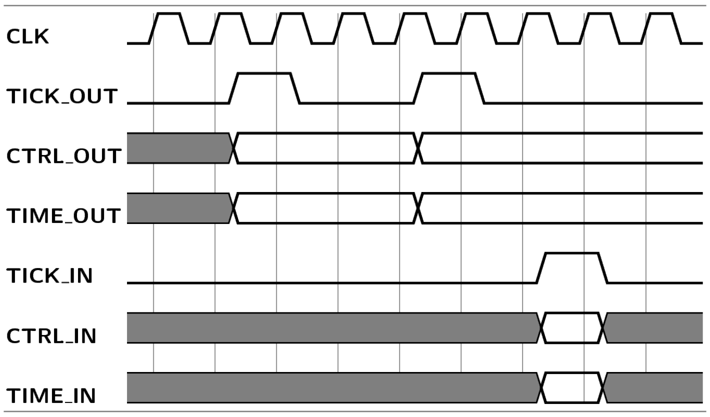

Figure 6: Timing of receiving/transmitting time-codes

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.

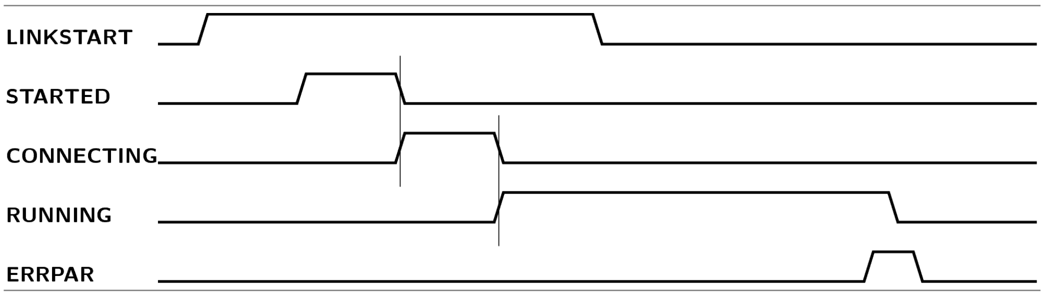

Figure 7: Sequence of link setup and status reporting (time not to scale).

5. AMBA interface

5.1. Overview

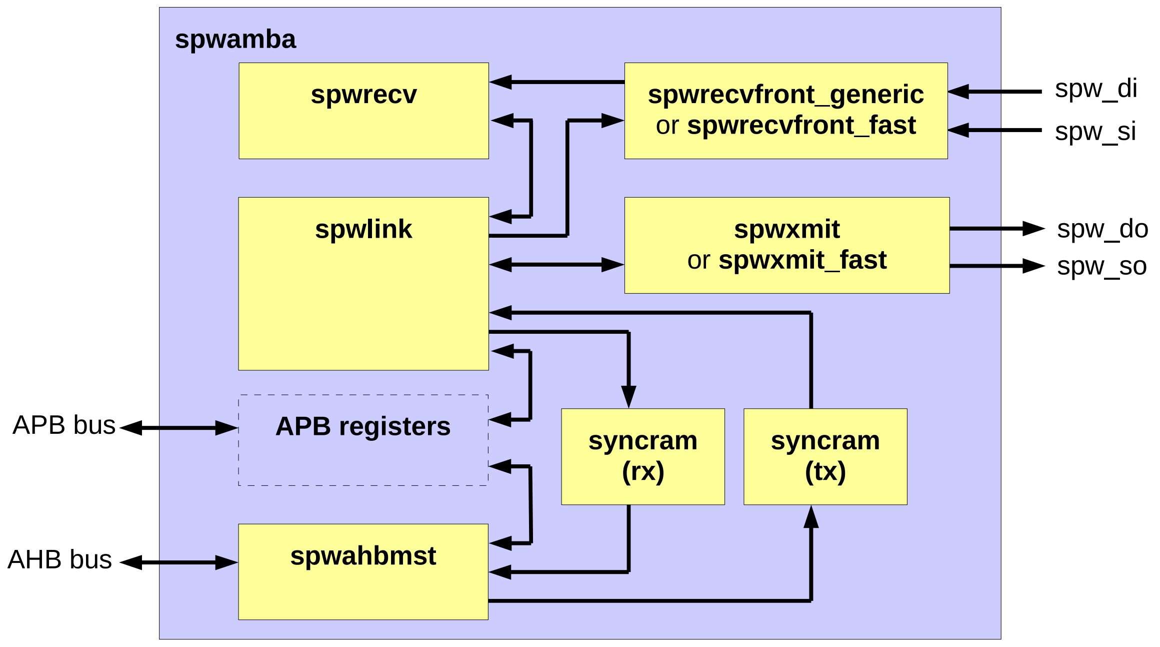

Figure 8: Block diagram of spwamba entity

The entity *spwamba* encapsulates a complete SpaceWire codec with an AMBA bus interface. The core is programmed through registers, accessible via an APB slave interface. SpaceWire data are transfered in DMA mode through an AHB master interface.

Spwamba is based on GRLIB, the system-on-chip library from Aeroflex Gaisler. As a result, it fits naturally in applications based on a LEON3 processor.

This entity depends on GRLIB. SpaceWire Light has been developed and tested with GRLIB version 1.2.2. The source code of GRLIB must be downloaded separately from the website of Aeroflex Gaisler [4] [5].

5.2. Operation

Link state management

The link is managed by a finite state machine as specified by the SpaceWire standard. The link state machine is controlled through three bits in the control register. When the *linkstart* bit is '1', 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 '1'. If *linkstart* is '0' but *autostart* is '1', the core waits passively until it detects a connection attempt from the other end of the link, before starting its own part of the handshake. If *linkdisable* is '1', the current link is shut down and no new link will be established (*linkdisable* overrides *linkstart*).

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.

Link status reporting

The status register reflects the current link status. If the link shuts down due to an error, the corresponding error bit is enabled in the status register. Error bits are sticky, i.e. remain set until explicitly cleared by software writing a '1'.

The core can optionally generate an interrupt on the APB bus whenever the link is established or shut down. These interrupts are enabled through a bit in the control register.

4 `http://www.gaisler.com/`
5 a copy of grlib-1.2.2 is archived at `http://jorisvr.nl/grlib/`

Receiving

Received data are transfered over the AHB bus to main memory. An internal receive FIFO holds received characters until the DMA engine is ready to transfer them over the AHB bus. 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 the *linkdisable* bit.

Transmitting

Outgoing data are transfered over the AHB bus from main memory to an internal FIFO. Characters are taken from the FIFO and transmitted on the SpaceWire link, provided that 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 transfered to the FIFO 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 the *linkdisable* bit.

Transmission bit rate

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 bit rate can be changed at run time by writing to the transmission clock scaler register. After reset, this register is initialized for a 10 Mbit bit rate. Link initialization is always done at 10 Mbit/s, regardless of the value in the scaler register.

Time-codes

Transmission of a time-code can be initiated in two ways: either through a pulse on the *tick_in* signal, or by writing to the time-code register. The *tick_in* signal is ignored unless it is enabled by setting a bit in the control register. The value of outgoing time-codes is determined by the time-code register. This value is automatically incremented after each transmitted time-code.

The last received time-code is reflected in the time-code register. The core can optionally generate an interrupt after each received time-code.

5.3. DMA interface

Incoming and outgoing packets are transfered over the AHB bus. These bus transfers are initiated by the *spwamba* core, i.e. the core is acting as an AHB bus master.

Two DMA channels operate on the AHB bus, one channel for RX transfers and one channel for TX transfers. Although these two channels share a single AHB master interface, they operate largely independent of each other. At any time, either one or both or none of the channels may be active. If both channels are active and ready to transfer data, the core alternates between channels, switching to the other channels after each maximum burst transfer.

Descriptor tables

Each DMA channel is controlled by its own descriptor table. The descriptor tables reside in main memory. The address of each table is written to a register in the *spwamba* core. The use of descriptor tables enables the core to perform a series of bus transfers without any need for software intervention.

The size of the descriptor tables is determined by the *desctablesize* generic, where each table entry takes 8 bytes. The first descriptor in a table must be aligned to the size of the entire table. For example, if *desctablesize* = 10, there are 1024 entries per table, thus the size of a table is 8192 bytes, therefore the address of the table must be aligned to 8192 bytes.

Once a DMA channel is started, it remains active as long as it finds descriptors with the *enabled* bit set. If the end of the table is reached, processing continues from the beginning of the table. Software may also set a *wrap* bit in a descriptor to indicate that the core should wrap back to the beginning of the table even before the end of the table is reached. If the core encounters a descriptor which is not enabled, it stops processing that channel. Software may restart the channel after setting up new descriptors.

Descriptor format

A descriptor consists of two 32-bit words. The first word specifies the size of the data fragment associated with the descriptor, as well as a set of flags. The second word specifies the address of the data fragment. This address must be aligned to 4 bytes.

The 16 least significant bits of the first word specify the size of the data fragment. For an enabled, not yet processed RX descriptor, this field contains the maximum number of bytes which may be stored in the associated data fragment (must be a multiple of 4). When an RX descriptor is completed, the core replaces this field with the actual number of bytes transfered. For an enabled, not yet processed TX descriptor, this field contains the number of bytes to be transfered. When a TX descriptor is completed, this field becomes undefined.

The *EN* bit of the flag word determines whether the descriptor is enabled. When the core reaches a disabled entry in a descriptor table, it stops processing that table. When the core completes a descriptor, it sets the *EN* bit to '0' and the *DONE* bit to '1'.

The *WR* bit may be set to wrap back to the beginning of the table before the end of the table is reached. When the core completes a descriptor with the *WR* bit set, it attempts to read the next descriptor from the first entry of the table.

When the core completes a descriptor with the *IE* bit set, it enables the RX-complete bit in the status register. If RX-completion interrupts are enabled, the core also generates an interrupt on the APB bus.

The *EOP* and *EEP* bits are used to indicate the end-of-packet marker received after the data fragment (RX) or to specify the end-of-packet marker to be sent after the data fragment (TX).

Descriptor flags (first word of each descriptor)

bits	description	
`15:0`	(RX, enabled) (RX, completed) (TX, enabled) (TX, completed)	SIZE: maximum number of bytes to store in this fragment (multiple of 4) LENGTH: number of bytes received LENGTH: number of bytes to transmit undefined
`16`	EN: '1' if this descriptor is enabled and not yet completed	
`17`	WR: '1' to wrap to the beginning of the descriptor table	
`18`	IE: '1' to generate interrupt and raise status flag when this descriptor is completed	
`19`	DONE: '1' if descriptor is completed	
`20`	(RX, completed) (TX, enabled)	EOP: '1' if received EOP after data in this fragment EOP: '1' to send EOP after data in this fragment
`21`	(RX, completed) (TX, enabled)	EEP: '1' if received EEP after data in this fragment EEP: '1' to send EEP after data in this fragment

Starting DMA transfers

To start a DMA channel, setup one or more entries in the descriptor table, write the address of the descriptor table to the channel's DMA address register, and start the channel by writing to the control register. The core will autonomously fetch descriptors, perform DMA transfers and update completed descriptors until it reaches a disabled descriptor in the table.

Software may monitor the progress of DMA transfers, either by reading the DMA address register or by testing the *DONE* bit of a pending descriptor. Alternatively, software may enable descriptor interrupts to get an interrupt when the core completes specific descriptors.

A descriptor must not be modified while its *EN* bit is set and the DMA channel is active. Such descriptors are “owned” by the *spwamba* core. When the core completes the descriptor, it will clear the *EN* bit and set the *DONE* bit, indicating that it has released that descriptor.

The descriptor table may be refilled at any time by writing new valid entries just after the last enabled entry. Care should be taken to set the *EN* bit only when the descriptor is completely valid. For example, write the address word of the descriptor before writing the flag word. After refilling one or more descriptor entries, the DMA channel should be restarted by writing to the control register. Restarting the DMA channel is recommended even if the channel was still active at the time of refilling, because it avoids the case where the channel reaches a disabled descriptor just before the new entry is written (if the channel is still active, the restart command is ignored).

RX transfers

While the RX channel is active, the core transfers incoming characters from the RX FIFO to main memory. If the FIFO becomes empty, the DMA transfer pauses until more characters arrive. The core completes a descriptor either when the requested number of bytes have been transfered or when an end-of-packet marker is encountered. The core then fetches the next descriptor and continues transferring data, unless the next descriptor is disabled in which case the RX channel stops.

An active RX descriptor is only completed when either the associated data fragment is completely filled or an end-of-packet marker is received. This implies that received characters may linger in an active RX descriptor indefinitely until the corresponding end-of-packet marker is received. In other words: data delivery is packet-based, not character-based.

If an incoming packet is larger than the number of bytes requested in the current descriptor, the remainder of the packet is transfered to the next descriptor. The size of incoming packets is thus not limited by the size of data fragments in the descriptor table, because large packets are automatically split over multiple descriptors.

If the size of an incoming packet exactly matches the number of bytes requested in an RX descriptor, the descriptor may be completed without any end-of-packet indication. The end-of-packet marker is then returned in the next descriptor, together with a zero-length data fragment. Thus in order to receive a complete packet in a single descriptor, the number of requested bytes must be strictly larger than the expected packet size.

If the last enabled descriptor has been completed and no further characters are received on the SpaceWire link, the DMA channel will remain active even though there are no more enabled descriptors. As soon as more characters are received, the core will notice the disabled descriptor and deactivate the channel. As a result, the RX channel will typically remain active even when no more data is expected from the SpaceWire link. Note that there is no harm in this situation, since the DMA channel does not consume bus cycles unless it actually transfers data.

TX transfers

While the TX channel is active, the core transfers data from main memory to the TX FIFO. If the TX FIFO becomes full, the DMA transfer pauses until there is again room in the FIFO. The core typically waits until the TX FIFO has room for a maximum length burst before starting a new bus transfer.

When the end of a data fragment is reached, an end-of-packet marker is added to the TX FIFO if one was specified in the descriptor. The descriptor is then completed and the next descriptor is fetched. Data transfer continues with the new descriptor, unless the new descriptor is disabled in which case the TX channel stops.

It is possible to split an outgoing packet over multiple subsequent TX descriptors. In this case only the last descriptor specifies an end-of-packet marker. It is acceptable for this last descriptor to have a zero-length data fragment.

AHB errors

If an AHB error occurs during a DMA transfer, the core stops both DMA channels and sets an error bit in the status register. The core will not perform any further AHB transfers after an AHB error has occurred. To recover from this state, a reset of the DMA engine or a reset of the entire *spwamba* core is required. A soft reset can be requested through the control register.

Cache and ordering considerations

When accessing descriptors from software, certain operations must be performed in a particular order. For example, when adding a descriptor to the descriptor table of an active DMA channel, the *EN* bit must be written after writing the data pointer but before writing the control register to restart DMA.

Putting these operations in the right order in source code does not guarantee correct ordering. Compilers tend to change the order of memory operations unless explicitly instructed not to do so (e.g. by marking such operations `volatile`). Some processors (but not the LEON3) may still reorder even volatile memory operations, unless explicitly prevented by a memory fence.

Cache may cause problems when reading descriptors and/or data which have been modified by the *spwamba* core through DMA. If the processor reads from its cache instead of from actual memory, it may see an old value which has not yet been affected by the DMA transfer. Some processors avoid this issue through *cache snooping*: selectively invalidating cache contents when another bus master writes to a cached region. In the LEON3, cache snooping is an optional feature. If snooping is not enabled in the processor configuration, a different mechanism must be used to avoid reading stale data from the cache. Options include programming the MMU for uncached access, or using LEON3-specific load instructions to bypass the cache.

5.4. Registers

The following registers are accessible through the APB slave interface. Unless otherwise specified, register bits are readable and writable and retain the last written value. Exceptions are read-only bits, sticky bits, auto-cleared bits, and auto-incrementing counters. After reset, all registers are initialized to zero, except for the transmission clock scaler which is initialized to 10 Mbit.

offset from APB base address	description
`0x00`	Control register
`0x04`	Status register
`0x08`	Transmission clock scaler
`0x0c`	Time-code register
`0x10`	Descriptor pointer for RX DMA
`0x14`	Descriptor pointer for TX DMA

Control register (offset 0x00)	
bits	**description**
0	Write '1' to reset SPWAMBA core (auto-clear).
1	Write '1' to reset DMA engines (auto-clear).
2	Link start signal. '1' = actively try to start a SpaceWire link.
3	Link autostart signal. '1' = start link after receiving NULL from other side.
4	Link disable signal. '1' = shut down current link and do not establish new link.
5	Allow time-code transmission through *tick_in* signal.
6	Write '1' to (re-)start RX DMA (auto-clear).
7	Write '1' to (re-)start TX DMA (auto-clear).
8	Write '1' to cancel running TX DMA and discard data from TX FIFO (auto-clear).
9	Enable interrupt on link up/down.
10	Enable interrupt on time code received.
11	Enable interrupt on completed RX descriptor with IE='1'.
12	Enable interrupt on completed TX descriptor with IE='1'.
13	Enable interrupt on RX packet received.
27:24	Value of *desctablesize* generic (read-only).

Status register (offset 0x04)	
bits	**description**
1:0	Link status: 0=off, 1=started, 2=connecting, 3=run (read-only).
2	Got disconnect error (sticky, write '1' to clear).
3	Got parity error (sticky, write '1' to clear).
4	Got escape error (sticky, write '1' to clear).
5	Got credit error (sticky, write '1' to clear).
6	RX DMA running (read-only).
7	TX DMA running (read-only).
8	AHB error occurred (sticky, reset DMA engine to clear).
9	unused
10	Received time-code (sticky, write '1' to clear).

11	Completed RX descriptor with IE='1' (sticky, write '1' to clear).
12	Completed TX descriptor with IE='1' (sticky, write '1' to clear).
13	Received packet (sticky, write '1' to clear).
14	RX buffer is empty and last packet has been completely transfered to RX DMA (read-only).

Transmission clock scaler (offset 0x08)

bits	**description**
7:0	Clock division factor minus 1. The actual TX bit rate is determined by (txclk frequency) / (scaler + 1). During the handshake phase, this register is ignored and the TX bit rate is forced to 10 Mbit. After reset, this register defaults to the scaler value needed for 10 Mbit.

Time-code register (offset 0x0c)

bits	**description**
5:0	Last received time-code value (read-only).
7:6	Control bits received with last time-code (read-only).
13:8	Time-code value to send on next *tick_in* (auto-increment).
15:14	Reserved, write as zero.
16	Write '1' to send a time-code immediately (auto-clear).

Descriptor pointer for RX DMA (offset 0x10)

bits	**description**
2:0	Reserved, write as zero.
desctablesize+2:3	Descriptor index (auto-increment).
31:*desctablesize*+3	Fixed address bits of descriptor table.

Descriptor pointer for TX DMA (offset 0x14)

bits	**description**
2:0	Reserved, write as zero.
desctablesize+2:3	Descriptor index (auto-increment).
31:*desctablesize*+3	Fixed address bits of descriptor table.

5.5. Configuration

The entity is configured through the following VHDL generics:

name	**description**	**type**	**default**
`tech`	Memory technology for FIFOs.	integer	inferred
`hindex`	AHB master index	integer	
`pindex`	APB slave index	integer	
`paddr`	Bits 19 to 8 of the APB address range.	integer	
`pmask`	Mask for APB address bits 19 to 8.	integer	`0xfff`
`pirq`	Interrupt request line used by SPWAMBA.	integer	
`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`	Size of the receive FIFO as the 2-logarithm of the number of words (1 word = 4 bytes).	integer, 6 .. 12	8 (1024 bytes)
`txfifosize_bits`	Size of the transmit FIFO as the 2-logarithm of the number of words (1 word = 4 bytes).	integer, 2 .. 12	8 (1024 bytes)
`desctablesize`	Size of the DMA descriptor tables as the 2-logarithm of the number of descriptors.	integer, 4 .. 14	10 (1024 entries)
`maxburst`	Maximum burst length as the 2-logarithm of the number of words.	integer, 1 .. 8	3 (8 words)

5.6. 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`).
`rstn`	I	Synchronous reset (active low).
`apbi`	I	APB slave input signals.
`apbo`	O	APB slave output signals.
`ahbi`	I	AHB master input signals.
`ahbo`	O	AHB master output signals.
`tick_in`	I	Pulse high for one clock cycle to request transmission of a time-code.
`tick_out`	O	High for one clock cycle if a time-code was just received.
`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.

6. Code organization

`rtl/vhdl/`	Synthesizable VHDL code.
`spwpkg.vhd`	VHDL package with entity declarations.
`spwstream.vhd`	Top-level entity for SpaceWire core with FIFO interface.
`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.
`spwambapkg.vhd`	VHDL package with entity declarations for AMBA interface
`spwamba.vhd`	Top-level entity for SpaceWire core with AMBA interface.
`spwahbmst.vhd`	AHB master for AMBA interface.
`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
`sim/spwamba_leon3`	Test bench for *spwamba*.
`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.
`syn/spwamba_gr-xc3s1500`	Test design for LEON3 + SPWAMBA on GR-XC3S1500 board.
`sw/spwamba_test`	C program for testing/simulation of LEON3 + SPWAMBA.
`sw/rtems_driver`	RTEMS device driver for LEON3 + SPWAMBA and test program.

7. Test bench

Three test benches are available. One test bench to test the SpaceWire link, and one test bench for each application interface. The test benches have been developed with GHDL[6] (version 0.30) but will presumably also work with other VHDL simulators.

spwlink_tb

The purpose of *spwlink_tb* is to test the receiver, transmitter and link state machine at the level of SpaceWire signaling. 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.

A makefile is provided to run this test bench in GHDL: `sim/ghdl/Makefile`. For example:

```
cd sim/ghdl
make test_spwlink
```

streamtest_tb

The second test bench, *streamtest_tb* is intended to test 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.

A makefile is provided to run this test bench in GHDL: `sim/ghdl/Makefile`. For example:

```
cd sim/ghdl
make test_streamtest
```

spwamba_tb

A test bench for *spwamba* is provided in the directory `sim/spwamba_leon3`. This test bench instantiates a minimal LEON3 system with a SpaceWire Light core. The simulated system includes 128 kByte of RAM memory. At the start of the simulation, a software image is loaded into this memory from the file `spwamba_test.srec`. The actual tests are implemented in the software image.

A makefile is provided to run the test bench. The test bench requires GRLIB 1.2.2. It may be necessary to modify the makefile to specify the location of the GRLIB directory. The following commands may be used to build and run the simulation:

```
cd sim/spwamba_test
make ghdl
make ghdl-launch
```

The same makefile may also be used to rebuild the software image. Source code for this image is in another directory: `sw/spwamba_test`. To build the test image, the LEON3 Bare-C (BCC) compiler system is required. The compiler is available from the Gaisler website as `sparc-elf-4.4.2-1.0.35`. The following commands may be used to build the test image:

```
cd sim/spwamba_test
make spwamba_test.srec
```

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

8. Synthesis guidelines

8.1. Platform support

The core has until now only been tested on Xilinx Spartan-3, Virtex-5 and Spartan-6, using Xilinx ISE 14.4.

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*.

8.2. 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.

8.3. 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 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 DATAPATHONLY;
TIMESPEC "TS_sys_to_rx" = FROM "clk" TO "rxclk" 10 ns DATAPATHONLY;
TIMESPEC "TS_tx_to_sys" = FROM "txclk" TO "clk" 8 ns DATAPATHONLY;
TIMESPEC "TS_sys_to_tx" = FROM "clk" TO "txclk" 8 ns DATAPATHONLY;
```

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).

8.4. I/O registers

To minimize skew between the data and strobe lines, it is important that *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.

Some FPGA families (including Xilinx Spartan) have an option to implement input and output flip-flops inside I/O elements. 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 flip-flops with the Xilinx command line tools, specify "`-pr b`" as an option to the mapper.
To enable I/O flip-flops in Xilinx ISE, got to Implement, Mapper, Process Options, and set "Pack I/O registers into IOBs" to "Inputs and Outputs".

8.5. Integration in GRLIB

The AMBA interface to SpaceWire Light depends on GRLIB, a VHDL library from Aeroflex Gaisler. Development and testing has been done with version GRLIB 1.2.2. The source code of GRLIB must be downloaded separately from the website of Aeroflex Gaisler [7] [8].

In the example design for LEON3 and SPWAMBA (`sim/spwamba_gr-xc3s1500`), the SpaceWire Light core is in an external directory, separate from the rest of GRLIB. The location of the GRLIB directory is specified in the makefile of the top-level design.

In real applications, it will probably be easier to integrate SpaceWire Light into GRLIB, in the same way that other cores are part of the library. This can be done as follows:

1. Create a new directory inside GRLIB: `lib/opencores/spacewirelight`.
   ```
   $ cd grlib-1.2.2
   $ mkdir lib/opencores/spacewirelight
   ```
2. Copy all VHDL files from `spacewire_light/rtl/vhdl` to the new directory `grlib/lib/opencores/spacewirelight`.
   ```
   $ cp ~/spacewirelight/rtl/vhdl/*.vhd lib/opencores/spacewirelight
   ```
3. Create a new text file `lib/opencores/spacewirelight/vhdlsyn.txt` containing the file names of the copied VHDL files.
   ```
   $ cd lib/opencores/spacewirelight
   $ echo *.vhd > vhdlsyn.txt
   ```
4. Edit the file `lib/opencores/dirs.txt` and add the word `spacewirelight` at the end of the line.

Once this is done, the SpaceWire Light core may be added to an existing LEON3 top-level design. Simply add "`use`" statements for the packages `opencores.spwpkg.all` and `opencores.spwambapkg.all` and add an instantiation of `spwamba`, properly attached to the APB and AHB controllers. The SpaceWire input/output ports need to be added to the top-level entity and to the constraints file. Also, if SpaceWire Light is used with multiple clock domains, timing constraints should be created as described above.

See also the example design `sim/spwamba_gr-xc3s1500/leon3mp.vhd`.

7 http://www.gaisler.com/
8 a copy of grlib-1.2.2 is archived at http://jorisvr.nl/grlib/

9. Software

The SPWAMBA core may be accessed from a C program on a LEON3 system. A sample program is available to test this mechanism. There is also a device driver for RTEMS.

9.1. Stand-alone test program

A minimal test program for the SPWAMBA core is available in `sw/spwamba_test`. This program runs under the LEON3 Bare-C environment. The program performs a fixed sequence of tests on the SPWAMBA core. It prints progress information to the console after each test step.

The program can be used either in simulation or on actual hardware. To use the program for simulation, it should be built with the makefile that belongs to the simulation. See §7 for more information about simulating SPWAMBA.

When running on real hardware, the test program assumes that a loopback cable is installed on the SpaceWire port, wiring TX pins to the corresponding RX pins. To use the program on hardware, it should be built with its own makefile. The LEON3 Bare-C (BCC) compiler system is required. This compiler is available from the Gaisler website as `sparc-elf-4.4.2-1.0.35`. Running `make` should produce the binary file `spwamba_test.dsu`. This file can be executed on a LEON3 board using `grmon`, the LEON3 debug monitor from Gaisler.

Note: the test program assumes that the LEON3 processor supports data cache snooping.

9.2. RTEMS driver

An RTEMS[9] device driver for the SPWAMBA core is available in `sw/rtems_driver`. The driver works under RTEMS 4.10 on LEON3 systems. It provides access to most features of the SPWAMBA core. The application interface to the driver is documented in the header file, `spacewirelight.h`.

`spwltest` is a test application for SPWAMBA, based on the RTEMS driver. The application interacts with the user through a simple menu interface. One of the features is a loopback test, where the application sends a sequence of packets through the SpaceWire link and expects to receive the same data sequence back. The loopback test can be used with direct loopback wires installed on the SpaceWire port (i.e. wires connecting DataIn to DataOut and StrobeIn to StrobeOut). Alternatively, the loopback test can be used with a remote SpaceWire device which is programmed to echo anything it receives.

The test application also has a passive loopback feature. In this mode, any data packets received from the SpaceWire link are echoed back to the SpaceWire link. This feature may be useful in combination with an active loopback test running on a remote SpaceWire device.

Note: Gaisler distributes its own modified version of RTEMS for the LEON3 (they sometimes call this RCC). The SpaceWire Light driver does not work with this Gaisler-specific version of RTEMS. The driver requires an unmodified RTEMS system from www.rtems.org.

9 Real-time embedded operating system; `http://www.rtems.org/`

10. Performance

10.1. 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)

10.2. Resource utilization and timing of spwstream interface

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 12.4.

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	166	343
Nr of LUTs	428	752
Nr of slices	248	459
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.

10.3. Resource utilization of spwamba interface

The resource utilization of *spwamba* may be estimated from the difference between two LEON3 designs, one with and one without SpaceWire Light. Both designs were synthesized for Xilinx XC3S1500 with ISE 12.4.

Design	LEON3 with SPWAMBA	LEON3 without SPWAMBA	Difference

Configuration	rximpl = impl_fast
	tximpl = impl_fast
	rxchunk = 4
	rxfifosize = 8
	txfifosize = 8
	desctablesize = 10		

Nr of flip-flops	7778	7075	703
Nr of LUTs	23647	21724	1923
Nr of slices	12294	11680	614
Nr of block RAMs	22	20	2
Nr of DCMs	3	2	1