RMAP Initiator: Product User Guide

Contents

Overview 3
Core Architecture 3
RMAP Command Controller 3
Example Command Byte Structure 5
Header & Data Interface 5
RMAP Reply Controller 7
Example Reply Byte Structure 7
Header & Data Interface 8
RMAP SpaceWire CoDec 9
Design Notes 9
Core Configuration 9
Core Interface 10
Parallel & FiFo Command/Reply Interface 10
Tx Parallel Interface 11
Submitting Path Addresses with Parallel Interfaces 11
Rx Parallel Interface 11
Clock & Reset Signals 12
(Parallel) Command Header Ports 12
Command Header Ports 12
Command Data ports 13
Command Error ports 13
(Parallel) Reply Header Ports 13
Reply Header ports 13
Reply Data ports 13
Reply Error ports 14
SpaceWire Status ports 14
SpaceWire Time Code ports 14
SpaceWire IO Ports 14
VHDL Packages 15
rmap_initiator_lib 15
SpaceWire_Sim_lib 15
RMAP Simulation Framework (VHDL) 15

RMAP Protected Types & Records 15
Core Example Design (Xilinx Vivado) 16
Getting Started .... **Error! Bookmark not defined.**
Behavioural Simulation 19

Figure 1: Simplified Core Architecture 3
Figure 2: Command Controller FSM 4
Figure 3: Standard RMAP Write Command Format 5
Figure 4: IP -Write Command Submission Format 5
Figure 5: Byte Submission Handshake 6
Figure 6: Tx Data Interface waveform 6
Figure 7: Reply Controller FSM 7
Figure 8: RMAP Read-Reply Byte Format 7
Figure 9: IP Read-Reply Output Format 8
Figure 10: Rx Data Ready/Valid Handshake 8
Figure 11:SpaceWire IO Configuration 10
Figure 12: Initiator Generics Example Settings 11
Figure 13: IP DDR Reg / IO Connections 15
Figure 14: SpaceWire Debug Signal Declaration 16
Figure 15: SpaceWire Debug in Testbench 16
Figure 16: SpaceWire Debug Sim Waveform 16
Figure 17: Testbench Architecture 17
Figure 18: IP Folder Structure 18
Figure 19: Example Design & Simulation Sources (Vivado) .... **Error! Bookmark not defined.**
Figure 20: Simulation Finished OKAY .... **Error! Bookmark not defined.**

Overview

The 4Links RMAP Initiator IP Core implements a fully open-source FPGA IP core for posting RMAP commands to a SpaceWire network. The core is pre-configured to include the 4Links SpaceWire IP Codec IP Core as standard. All supporting VHDL libraries and IP are open-sourced and accessible for modification.

The core complies with the behaviour for an RMAP initiator as outlined in ECSS-E-ST-50-52C. The IP core is intended to be used with the 4Links RMAP Target and 4Links RMAP Router IP Core(s). All of which are open-source, free to use, SpaceWire IP provided by 4Links.

Core Architecture

The RMAP initiator is comprised of two distinct parts, a Command Controller (Tx) and a Reply Controller (Rx). The Rx and Tx sides of the core function independently of one another. Allowing a command to be posted as a reply is received and vice-versa. Each controller has its own CRC logic for implementing the RMAP compliant CRC algorithm. CRC bytes are automatically calculated and checked by the initiator IP core. CRC errors are reported to the user application as they occur.

Figure 1 shows a simplified diagram of the IP core architecture. Note the 4Links SpaceWire CoDec IP core is already instantiated within the RMAP Initiator.

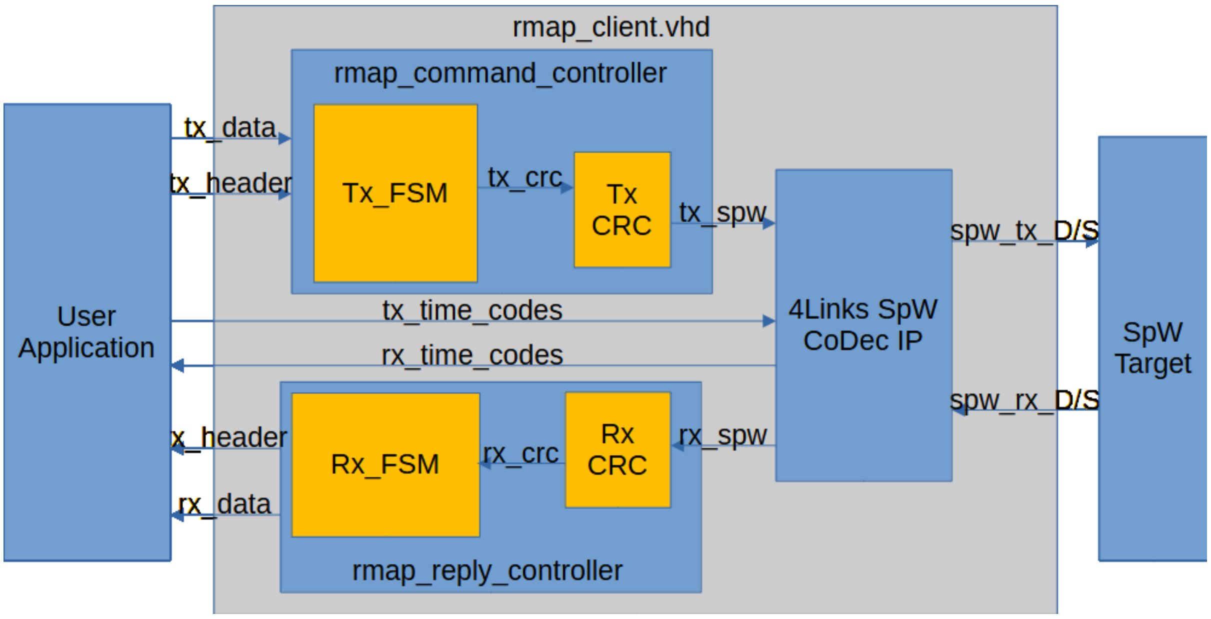


Figure 1: Simplified Core Architecture

RMAP Command Controller

RMAP header and data bytes are checked before being sent across the SpaceWire link. Minimal buffering is used to prevent SpaceWire link hang-ups when using Wormhole routing. The CRC calculator performs CRC calculations automatically as data is streamed across the SpaceWire link. The CRC implementation meets the requirements as laid out by ECSS-E-ST-50-52C. Checking includes early EEP/EOP detection.

CRC Bytes and EOPs are automatically appended to the RMAP command by the Command Controller. The FSM for the command controller contains an error handler for unexpected behaviours. When in an error state, the error must be acknowledged by the host application before the Command Controller will recover to its initialized state. A list of error codes is presented in this document.

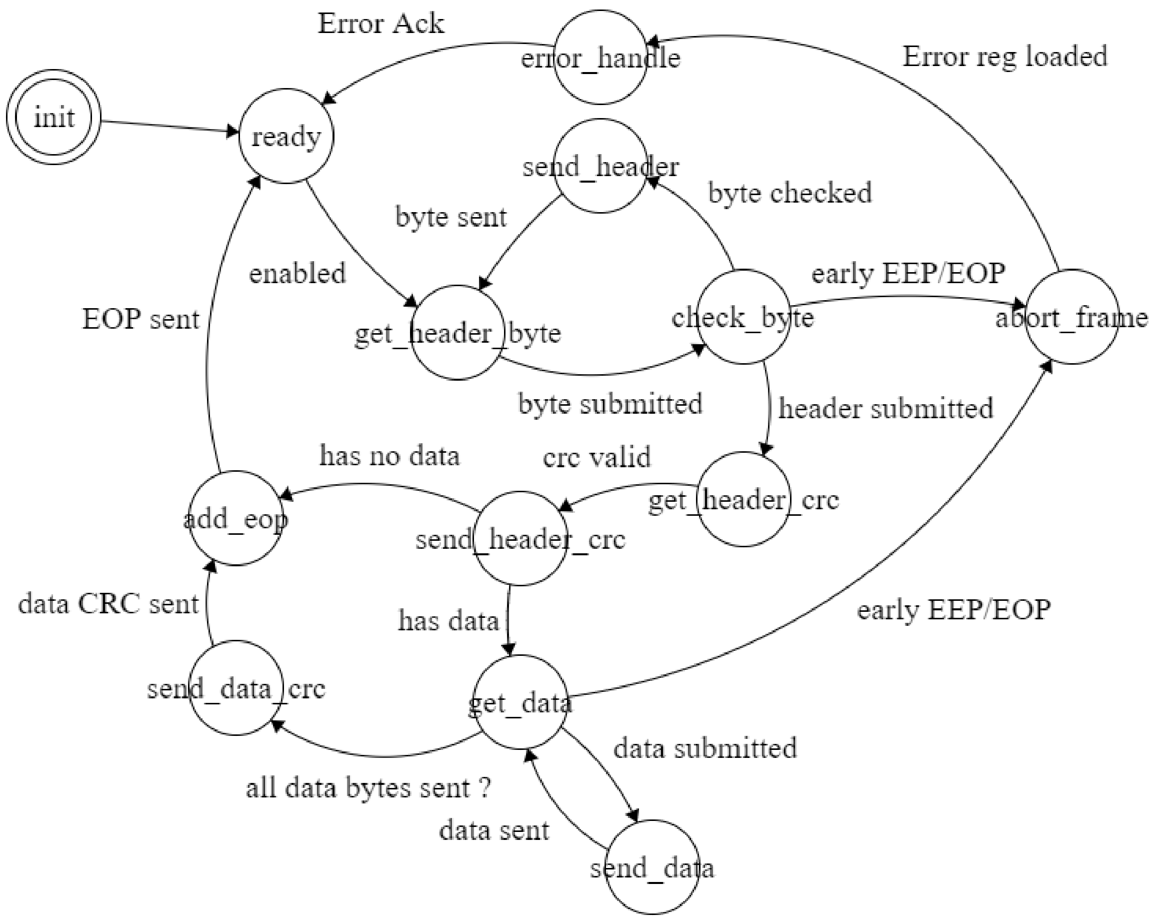
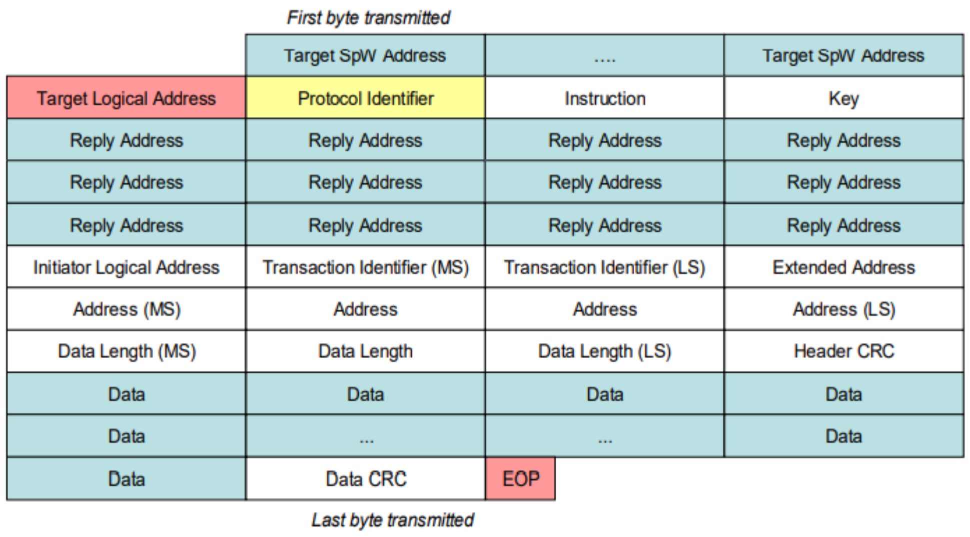


Figure 2: Command Controller FSM

In the case of an early EEP/EOP error state, if path addresses have already been submitted to the SpaceWire link, the EEP/EOP will be pushed and transmitted across the SpaceWire link. This ensures that any routers in the SpaceWire network will not lock up.

Example Command Byte Structure

Figure 3 contains the standard format for an RMAP write command. This is lifted from the ECSS-E-ST-50-52C RMAP specification.



First byte transmitted

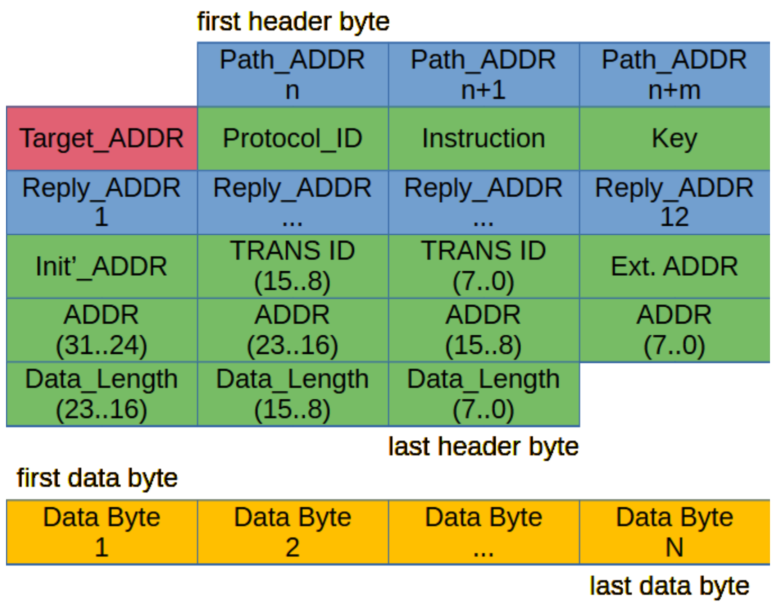
	Target SpW Address	....	Target SpW Address
Target Logical Address	Protocol Identifier	Instruction	Key
Reply Address	Reply Address	Reply Address	Reply Address
Reply Address	Reply Address	Reply Address	Reply Address
Reply Address	Reply Address	Reply Address	Reply Address
Initiator Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Extended Address
Address (MS)	Address	Address	Address (LS)
Data Length (MS)	Data Length	Data Length (LS)	Header CRC
Data	Data	Data	Data
Data	...	...	Data
Data	Data CRC	EOP	

Last byte transmitted

Figure 3: Standard RMAP Write Command Format

Below Figure 4 shows the submission order for an RMAP write command to the 4Links IP. Note that no Header/Data CRCs or EOP are required.

If using reply addressing and the number of reply addresses is not divisible by 4. Then the user application must pad the remaining reply address bytes with 0x00. Up to 12 reply addresses can be submitted.



first header byte

	Path_ADDR n	Path_ADDR n+1	Path_ADDR n+m
Target_ADDR	Protocol_ID	Instruction	Key
Reply_ADDR 1	Reply_ADDR ...	Reply_ADDR ...	Reply_ADDR 12
Init'_ADDR	TRANS ID (15..8)	TRANS ID (7..0)	Ext. ADDR
ADDR (31..24)	ADDR (23..16)	ADDR (15..8)	ADDR (7..0)
Data_Length (23..16)	Data_Length (15..8)	Data_Length (7..0)	

last header byte

first data byte

Data Byte 1	Data Byte 2	Data Byte ...	Data Byte N

last data byte

Figure 4: IP -Write Command Submission Format

Header & Data Interface

Submitting bytes to the RMAP Command interface is achieved using a ready/valid handshake. The ready signal is asserted by the IP core when the IP core is ready to receive data. The valid signal is asserted by the user application when valid data is presented on the interface. Data submission occurs only when both ready and valid signal are asserted. These actions are aligned with the rising edge of the positive system clock. Figure 5 shows an example of this handshake where the byte “0x6C” is submitted to the header interface.

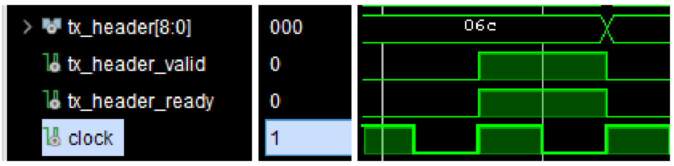


Figure 5: Byte Submission Handshake

When sending a command with a data payload. Once all header bytes have been sent, the data interface is used to transmit data bytes across the SpaceWire link (see Figure 6). As with the header bytes, a ready/valid handshake is used to load new data bytes onto the interface. Data bytes are checked for EEP/EOPs before transmission across the SpaceWire link.

The number of data bytes to be transmitted is set by the “Data_Length” field in the RMAP header. Injecting an EEP/EOP onto the data byte interface will force the RMAP initiator to abort the current RMAP frame and generate an error. The error must be acknowledged by the user application before the RMAP initiator can recover.

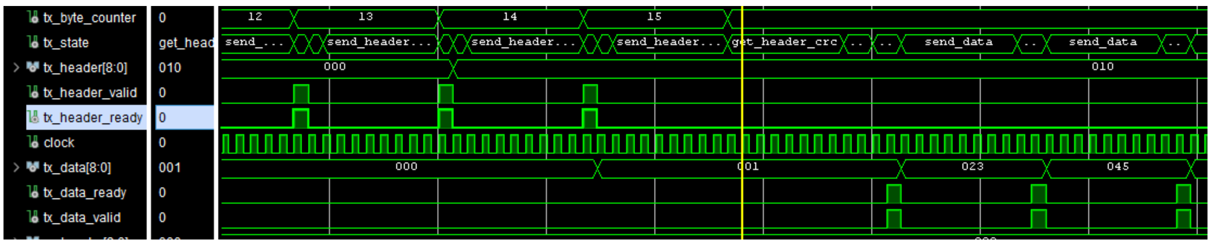


Figure 6: Tx Data Interface waveform

RMAP Reply Controller

The (Rx) Reply Controller functions independently to the RMAP initiator (Tx) Command Controller. This allows for fully asynchronous Tx/Rx operation. A simplified state-machine diagram for the Reply Controller can be found in Figure 7 below.

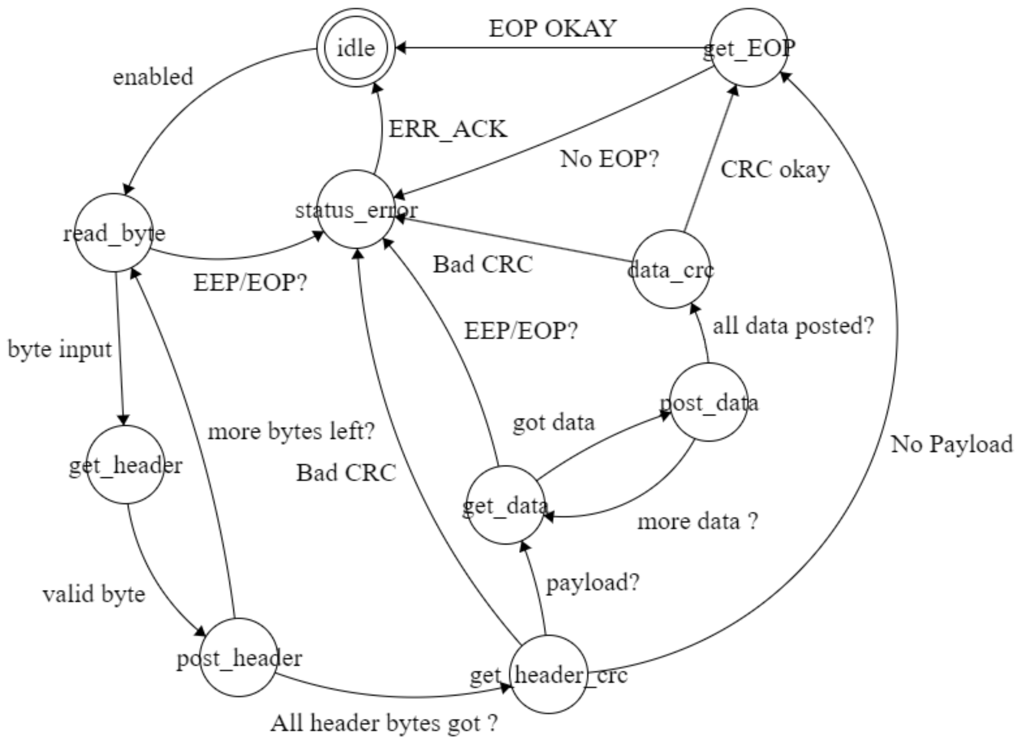
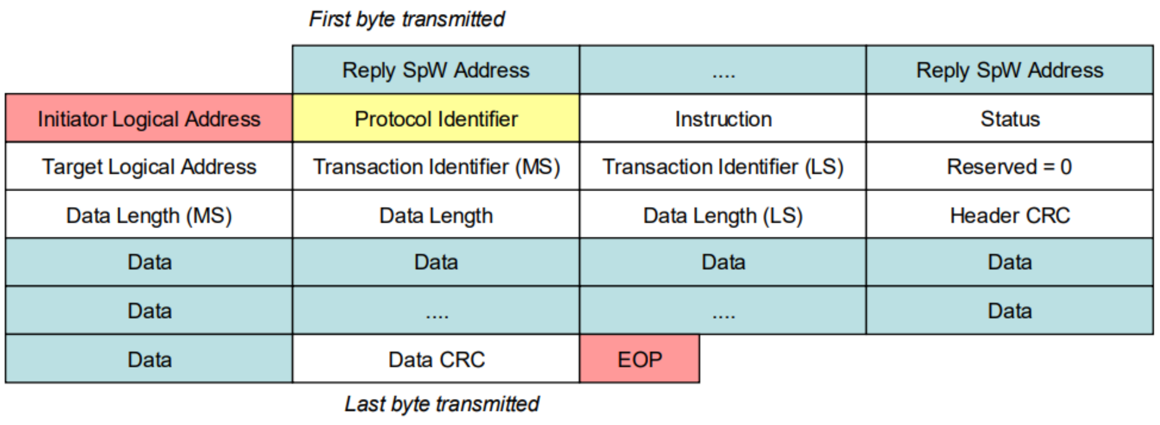


Figure 7: Reply Controller FSM

Example Reply Byte Structure

Figure 8 contains the RMAP Read Reply format. The RMAP Reply Controller complies with the RMAP ECSS-E-ST-50-52C specification.



First byte transmitted

	Reply SpW Address	....	Reply SpW Address
Initiator Logical Address	Protocol Identifier	Instruction	Status
Target Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Reserved = 0
Data Length (MS)	Data Length	Data Length (LS)	Header CRC
Data	Data	Data	Data
Data	....	....	Data
Data	Data CRC	EOP	

Last byte transmitted

Figure 8: RMAP Read-Reply Byte Format

The RMAP Reply Controller Strips off CRC, EOP and Reserved bytes (if present) when pushing Header/Data bytes to the user application. See Figure 9 for an example output of a Read-Reply frame.

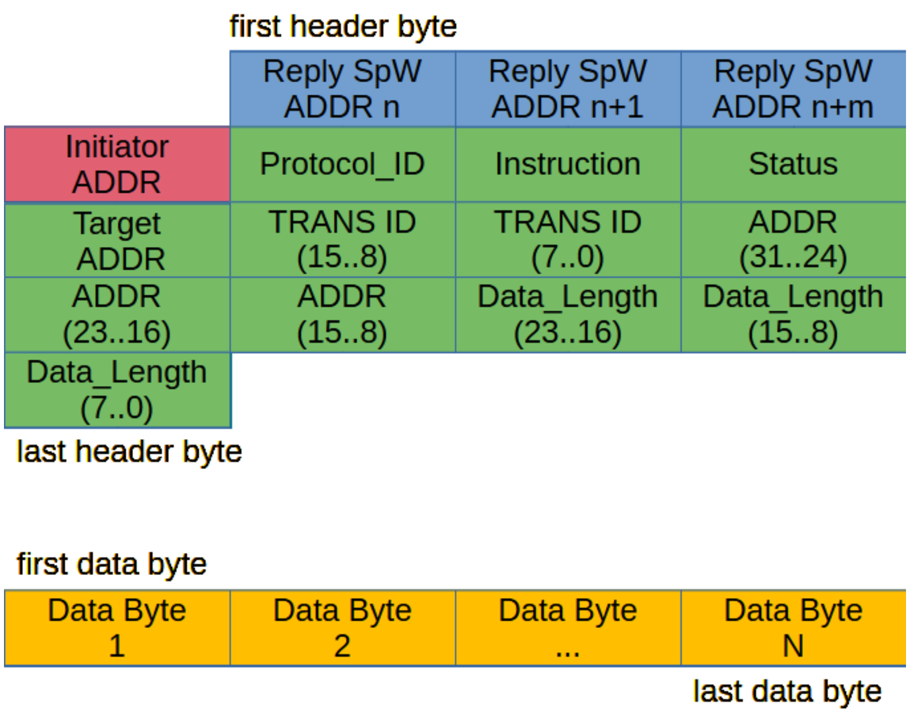


Figure 9: IP Read-Reply Output Format

Like the Command Controller, the Reply Controller uses separate header byte and data byte interfaces. Header and Data bytes are immediately pushed to the relevant interface. The RMAP Initiator IP does not perform any buffering of Reply bytes. If the user application interface is too slow, then Reply bytes could be dropped.

It is recommended that any application built around the Rx Header/Data interface uses its own dedicated FiFo Buffer logic to prevent data loss.

Header & Data Interface

Like the Command Controller, Header and Data bytes have their own interfaces. Once all header bytes have been received, if the reply contains a data payload, then the data bytes will be pushed to the Data interface.

Both Header and Data interfaces use a Ready/Valid handshake for exchanging header/data bytes with a user application. The *valid* signal is asserted by the RMAP initiator IP when a valid byte is present on the interface. The user application should assert the *ready* signal when it is ready to receive data. Figure 10 shows an example where the data byte 0x03 is transmitted, on the rx_data interface, to a user application.

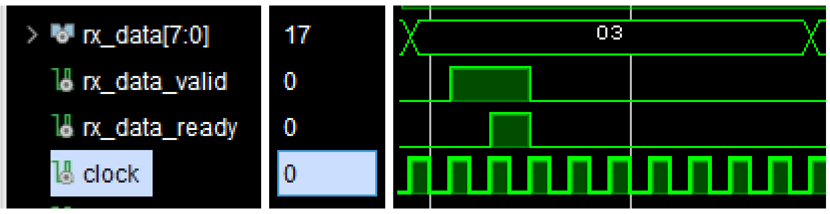


Figure 10: Rx Data Ready/Valid Handshake

RMAP SpaceWire CoDec

The RMAP Initiator is configured to use a generic wrapper version of the 4Links SpaceWire CoDec IP. It is recommended that “custom” mode is used. DDR registers/IO buffers can then be connected in the top-level design. This removes the need to modify the RMAP wrapper files, although that can be done if required.

The “single” and “diff” modes contain processes that model DDR inputs and LVDS IO buffers. Allowing for behavioural simulation without the need to modify the wrapper file.

If requiring a SpaceWire connection of >250Mb, please contact 4Links and we can provide a high-speed SpaceWire CoDec optimized for >350Mb operation. Achieving bitrates beyond 200Mb will greatly depend on your design size, architecture and FPGA technology.

Design Notes

This section contains operational and design notes which may prove useful when debugging designs with the RMAP Initiator IP.

- When using Ready/Valid handshakes, note that the *Valid* signal direction will always align with the *Data* signal direction. The *Ready* signal will always oppose the Data signal direction.

- The RMAP Initiator Command Controller is designed to saturate the SpaceWire Tx link during operation. This minimizes the number of NULL packets on the SpaceWire link and improves the true bitrate of the transaction. During large transactions this can lead to temporary stalls of the SpaceWire CoDec whilst it receives new FCTs. These stalls do not reduce the speed of data transmission, they simply show that the link is fully saturated.

- The RMAP Initiator Reply Controller does not buffer any Reply bytes. Therefore, data can be lost if the user application does not read the existing header/data byte before a new one is received. A buffer is not provided by default to facilitate higher bitrates.

- Bitrate is tied to the Input clock speed of the IP, such that, for a 100Mb connection, a clock frequency of 100MHz is required. Maximum achievable bitrate will depend on your target FPGA technology and size of your application. By default, the RMAP Initiator IP comes with 4Links Standard Open-Source SpaceWire CoDec IP Core. Higher-speed options are available on request.

Core Configuration

The IP core can be configured using VHDL Generics (parameters if using Verilog instantiation). RMAP initiator Generics are as follows:

NAME	TYPE	RANGE	DESC
g_clock_freq	REAL	>2_000_000.0	Clock Frequency of SpW
g_tx_fifo_size	INTEGER	>8	Tx FiFo Depth (SpW)
g_rx_fifo_size	INTEGER	>8	Rx FiFo Depth (SpW)
g_tx_fifo_interface	BOOLEAN	true/false	Generate Tx FiFo Core interface
g_rx_fifo_interface	BOOLEAN	true/false	Generate Rx FiFo Core interface
g_mode	STRING	“diff”, “single”, “custom”	IO mode of SpW IP

More information about these configuration constants can be found in the 4Links SpaceWire CoDec IP User Guide. See below how to configure the SpaceWire IO in “custom” mode.

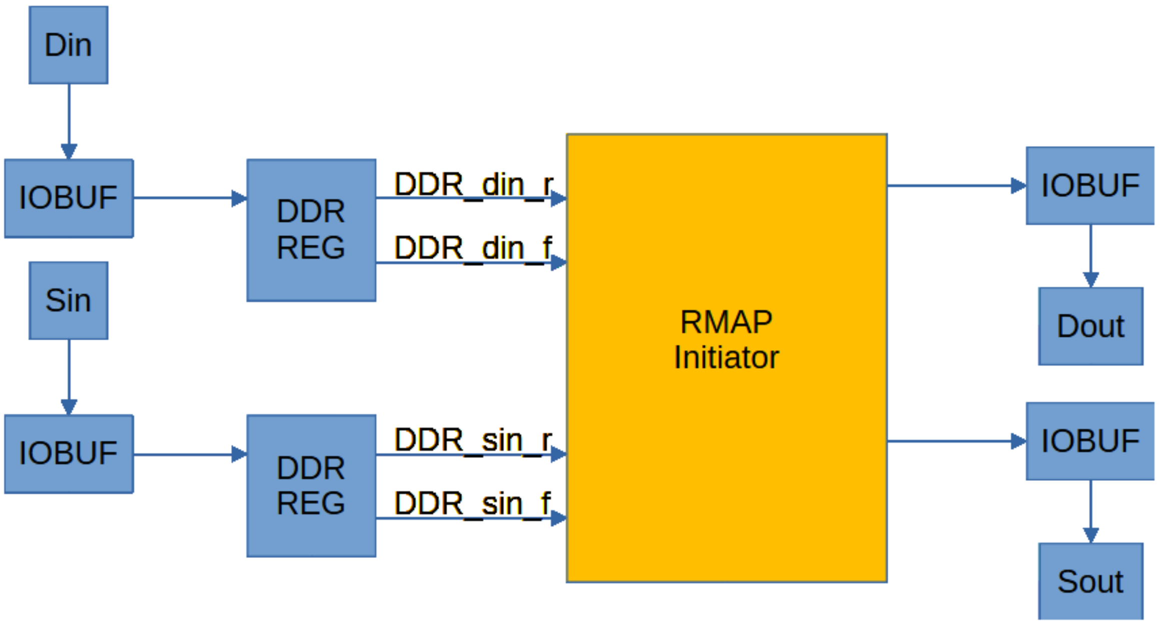


Figure 11:SpaceWire IO Configuration

Core Interface

For compatibility, use the *rmap_initiator_top_level* entity. Some ports on the *rmap_initiator* entity use record types. This can cause issues if instantiating directly into a System Verilog/Verilog project. The Top-level wrapper entity breaks the record connections out to use std_logic/vector IO. Using the *rmap_initiator_top_level* is recommended to avoid toolchain issues when switching between multiple HDLs within a single project.

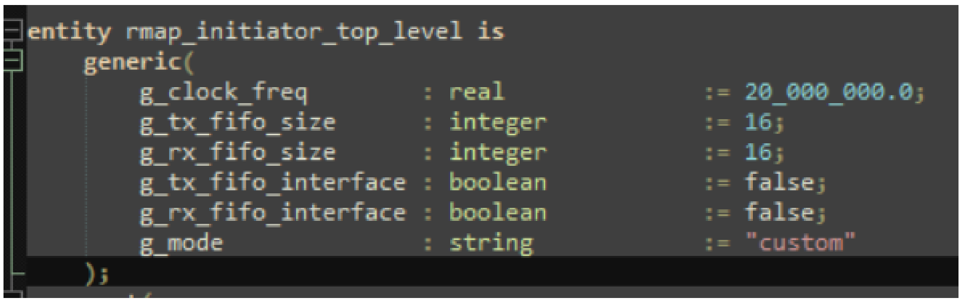
The Boolean-type interface ports on the SpaceWire CoDec IP are converted to use std_logic type ports for better design integration.

Parallel & FiFo Command/Reply Interface

The generics ***g_tx_fifo_interface*** and ***g_rx_fifo_interface*** allows selection between a parallel or FiFo-style interface for Tx/Rx header information. The FiFo-Style interface may be preferred when streaming RMAP command data to/from a processor and/or device memory. The FiFo interface also uses fewer device resources than the parallel interface.

The Parallel interface may be more useful for integration into RTL-Based RMAP controller designs and/or instances where RMAP header bytes remain constant between transactions.

When using parallel interface(s), the Command/Reply RMAP Data bytes still use a fifo-style write/read interface. The only difference is the header information for commands and replies.



```
entity rmap_initiator_top_level is
    generic(
        g_clock_freq       : real              := 20_000_000.0;
        g_tx_fifo_size     : integer           := 16;
        g_rx_fifo_size     : integer           := 16;
        g_tx_fifo_interface : boolean          := false;
        g_rx_fifo_interface : boolean          := false;
        g_mode             : string            := "custom"
    );
```

Figure 12: Initiator Generics Example Settings

Tx Parallel Interface

The Tx (command) parallel interface uses the ***tx_header_ready*** and ***tx_header_valid*** signals to submit header information to the command controller.

When the handshake is asserted, all configured header information is transmitted in the correct order across the SpaceWire link. Once all header information is sent, if required, the FiFo data interface will be used to submit required data bytes to the command controller.

The RMAP data port uses the same fifo-based submission regardless of core channel FIFO configuration.

Submitting Path Addresses with Parallel Interfaces

When using the Parallel Tx interface, path addresses should be submitted on the ***tx_logical_address*** interface one at a time. Once an address with value 0xFE is submitted on the ***tx_logical_address*** port, the Command controller will start transmitting all the configured header bytes as normal.

All header bytes should be configured before the address byte containing the default logical address (0xFE) is submitted to the command controller.

Rx Parallel Interface

SpaceWire path addresses are pushed to the ***rx_init_log_addr*** port until a logical address byte (0xFE) is received. Once the logical address byte is received, the rest of the header information will be obtained from the incoming RMAP reply.

The ***rx_header ready*** and ***rx_header_valid*** handshake is used to read the full RMAP header. If receiving path address bytes, the path address bytes must be acknowledged before the next header byte can be read in, until the logical address 0xFE is received.

Once this occurs, the Reply Controller will read the full header and push to the parallel interface once it has been obtained. If required, after the header has been received and acknowledged, incoming data bytes will be pushed to the ***rx_data*** interface. This interface uses the same ready/valid handshake in both Fifo and Parallel IO operating modes.

*** note ** acknowledgement refers to the assertion of the read/valid handshake between interface ports.*

Clock & Reset Signals

NAME	Direction	TYPE	WIDTH	Desc
clock	in	STD_LOGIC	--	Positive edge clock input
clock_b	in	STD_LOGIC	--	Negative edge clock input
rst_in	In	STD_LOGIC	--	Synchronous reset (active high)
enable	in	STD_LOGIC	--	Enable (active high)

The *clock* and *clock_b* signals should be produced using a MMCM/PLL/DLL. *clock_b* should be 180° out of phase from *clock*. All interface transactions to/from the RMAP Ip Core occur with respect to the rising edge of *clock*.

(Parallel) Command Header Ports

NAME	Direction	TYPE	WIDTH	Desc
tx_logical_address	in	STD_LOGIC_VECTOR	8	Target logical address
tx_protocol_id	in	STD_LOGIC_VECTOR	8	RMAP Protocol ID
tx_instruction	in	STD_LOGIC_VECTOR	8	Instruction Byte
tx_Key	in	STD_LOGIC_VECTOR	8	Target Key
tx_reply_addresses	in	BYTE_ARRAY	12	Reply addresses (up to 12)
tx_init_log_addr	in	STD_LOGIC_VECTOR	8	Initiator Logical Address
tx_Tranaction_ID	in	STD_LOGIC_VECTOR	16	RMAP Transaction ID
tx_Address	in	STD_LOGIC_VECTOR	40	Memory Address
tx_Data_Length	in	STD_LOGIC_VECTOR	24	Length of Data payload
Tx_header_valid	in	STD_LOGIC	--	Assert when header byte is valid
Tx_header_ready	out	STD_LOGIC	--	Asserted when logic is ready

Command Header Ports

NAME	Direction	TYPE	WIDTH	Desc
Tx_assert_path	in	STD_LOGIC	--	
Tx_assert_char	in	STD_LOGIC	--	Assert to send SpW Char
Tx_header	in	STD_LOGIC_VECTOR	8	Header byte to transmit
Tx_header_valid	in	STD_LOGIC	--	Assert when header byte is valid
Tx_header_ready	out	STD_LOGIC	--	Asserted when logic is ready

Tx_assert_path can be used to bypass a logical address and send it as a SpaceWire path address. Under normal operation, if a path address, the first SpaceWire address must be <32. If for whatever reason this is not the case, asserting "tx_assert_path" during the transaction will force the RMAP IP to send the address as a path address. Tx_assert_path is optional and is not required if the initial path address value is <32.

Tx_assert_char is used to send a SpaceWire control character. Asserting this high on a transaction will submit a SpaceWire character. The lower 2 bits of the Tx_header/Tx_data interfaces are used to assign the char value. "0x02 = EOP" etc.

Command Data ports

NAME	Direction	TYPE	WIDTH	Desc
Tx_assert_char	in	STD_LOGIC	--	Assert to send SpW Char
Tx_data	in	STD_LOGIC_VECTOR	8	Data byte to transmit
Tx_data_valid	in	STD_LOGIC	--	Assert when data byte is valid
Tx_data_ready	out	STD_LOGIC	--	Asserted when logic is ready

Command Error ports

NAME	Direction	TYPE	WIDTH	Desc
Tx_error	out	STD_LOGIC_VECTOR	8	Output Error Code
Tx_error_valid	out	STD_LOGIC	--	Error Code is valid
Tx_error_ready	in	STD_LOGIC	--	Assert to read Error Code

(Parallel) Reply Header Ports

NAME	Direction	TYPE	WIDTH	Desc
rx_init_log_addr	out	STD_LOGIC_VECTOR	8	Initiator Logical Address
rx_protocol_id	out	STD_LOGIC_VECTOR	8	RMAP Protocol ID
rx_instruction	out	STD_LOGIC_VECTOR	8	RMAP Instruction Byte
rx_Status	out	STD_LOGIC_VECTOR	8	RMAP Status Byte
rx_target_log_addr	out	STD_LOGIC_VECTOR	8	Target Logical Address
rx_Tranaction_ID	out	STD_LOGIC_VECTOR	16	RMAP Transaction ID
rx_Data_Length	out	STD_LOGIC_VECTOR	24	Length of Data Payload
crc_good	out	STD_LOGIC	--	Asserted when CRC is OKAY
Rx_header_valid	out	STD_LOGIC	--	High when header byte to read
Rx_header_ready	in	STD_LOGIC	--	Assert when ready for header byte

Reply Header ports

NAME	Direction	TYPE	WIDTH	Desc
Rx_assert_char	out	STD_LOGIC	--	Asserted when byte is SpW Char
Rx_header	out	STD_LOGIC_VECTOR	8	Received header byte
Rx_header_valid	out	STD_LOGIC	--	High when header byte to read
Rx_header_ready	in	STD_LOGIC	--	Assert when ready for header byte

Like Tx_assert_char, the *Rx_assert_char* port applies to both Header and Data bytes. This signal will be asserted when the current header/data byte to be read is a SpaceWire character.

EEPs/EOPs will not be pushed to the Header/Data interfaces, instead premature EOPs or any EEPs will generate a reply Error and must be acknowledged by the user application.

Reply Data ports

NAME	Direction	TYPE	WIDTH	Desc
Rx_assert_char	out	STD_LOGIC	--	Asserted when byte is SpW Char
Rx_data	out	STD_LOGIC_VECTOR	8	Received data byte
Rx_data_valid	out	STD_LOGIC	--	High when data byte to read
Rx_datar_ready	in	STD_LOGIC	--	Assert when ready for data byte

Reply Error ports

NAME	Direction	TYPE	WIDTH	Desc
Rx_error	out	STD_LOGIC_VECTOR	8	Output Error Code
Rx_error_valid	out	STD_LOGIC	--	Error Code is valid
Rx_error_ready	in	STD_LOGIC	--	Assert to read Error Code

SpaceWire Status ports

See 4Links SpaceWire CoDec IP User Guide.

SpaceWire Time Code ports

NAME	Direction	TYPE	WIDTH	Desc
Tx_time	in	STD_LOGIC_VECTOR	8	Timecode to send
Tx_time_valid	in	STD_LOGIC	--	Assert to send timecode
Tx_time_ready	out	STD_LOGIC	--	Asserted when ready for timecode
Rx_time	out	STD_LOGIC_VECTOR	8	Timecode received
Rx_time_valid	out	STD_LOGIC	--	Asserted when valid timecode
Rx_time_ready	in	STD_LOGIC	--	Assert to read timecode

SpaceWire IO Ports

NAME	Direction	TYPE	WIDTH	Desc
DDR_din_r	out	STD_LOGIC	--	Data In Rising Edge Sampled
DDR_din_f	out	STD_LOGIC	--	Data in Falling Edge Sampled
DDR_sin_r	out	STD_LOGIC	--	Strobe in Rising Edge Sampled
DDR_sin_f	out	STD_LOGIC	--	Strobe in Falling Edge Sampled
SDR_Dout	in	STD_LOGIC	--	Data output (Rising Edge Clock)
SDR_Din	in	STD_LOGIC	--	Strobe output (Rising Edge Clock)

DDR registers and IO buffers should be added outside of the RMAP initiator architecture (use g_mode = “custom”). See SpaceWire IP User guide for more detail.

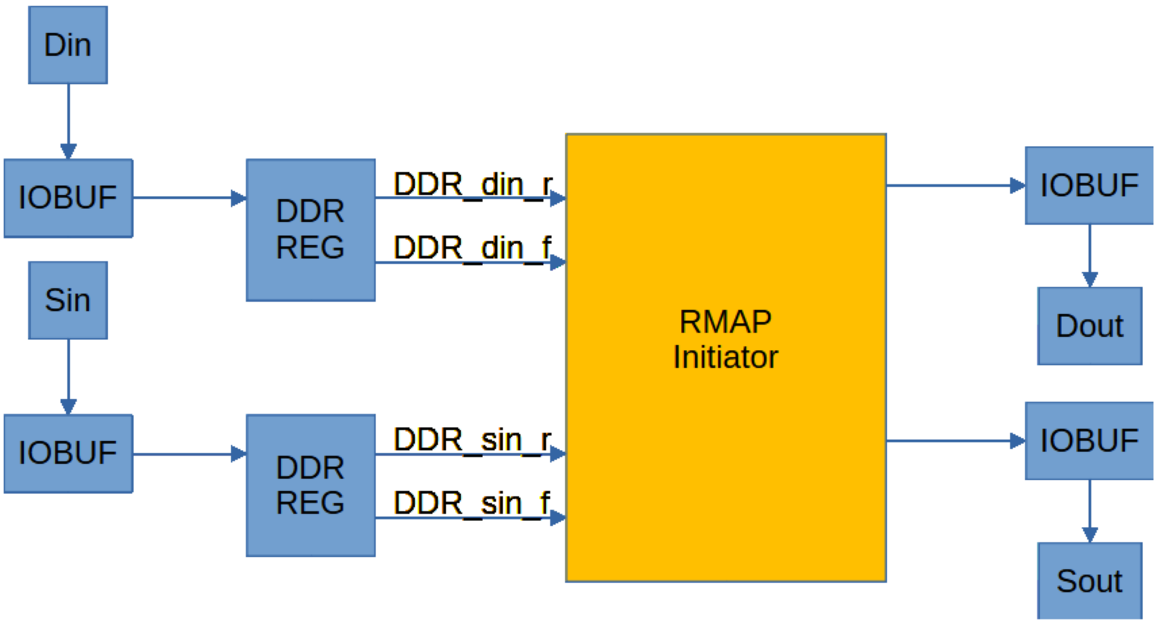


Figure 13: IP DDR Reg / IO Connections

VHDL Packages

Packages covered here are used to build and compile the 4Links RMAP Client IP.

rmap_initiator_lib

This package is used to create the RMAP Initiator IP. It contains records for simplifying interface connections between IP submodules. As well as useful types and functions. This package is required to build the RMAP initiator IP.

SpaceWire_Sim_lib

This package contains simulation procedures, types and constants to create a verification environment for 4Links RMAP IP(s). This package relies on interfaces and data types created in the *rmap_initiator_lib* package. This package is used by the testbench *rmap_initiator_tb* to construct a simple verification environment.

RMAP Simulation Framework (VHDL)

In order to ensure that designs using the 4Links RMAP Initiator IP are stable, a simulation environment with good functional coverage is required. Your approach to verification will depend on your specific application. With that in mind, the *SpaceWire_Sim_lib* package can be used to start developing your VHDL testbenches.

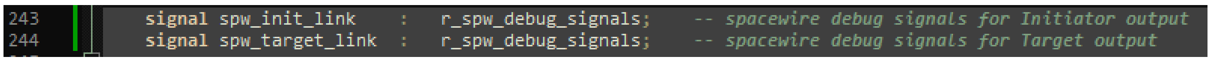
An example testbench can be found in the */Sim/* directory in the IP root folder.

RMAP Protected Types & Records

RMAP Command frames are set using protected types. The protected type *t_rmap_command* on line 214 of *SpaceWire_Sim_lib* is used to load and modify data buffers containing RMAP command data.

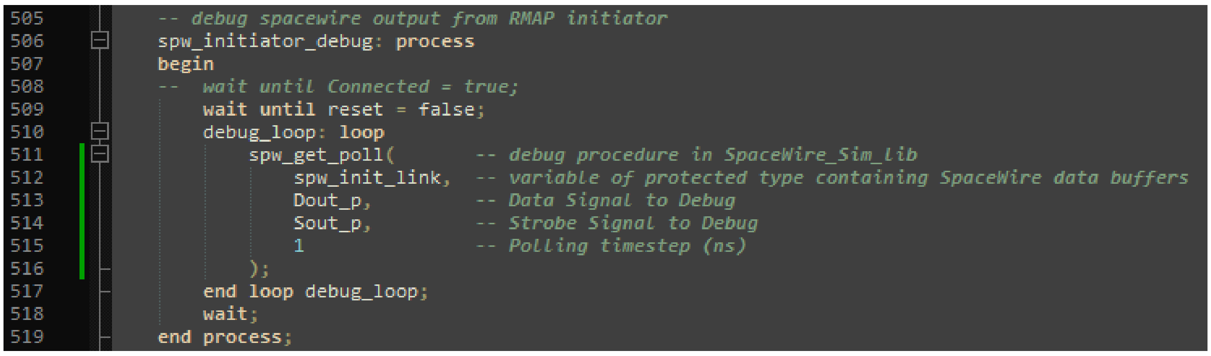
Protected types are used to allow for safe access of data by multiple processes and to create a class-like approach to verification. Where multiple related signals can be grouped to speed up verification.

The *spw_get_poll* procedure uses operator overloading to implement three different versions of the procedure. These allow for signal, record and protected type data outputs. The *rmap_init_tb* testbench uses record types to display SpaceWire debug information (see Figure 15, Figure 15 and Figure 16).



```
    signal spw_init_link    :   r_spw_debug_signals;    -- spacewire debug signals for Initiator output
    signal spw_target_link  :   r_spw_debug_signals;    -- spacewire debug signals for Target output
```

Figure 14: SpaceWire Debug Signal Declaration



```
    -- debug spacewire output from RMAP initiator
    spw_initiator_debug: process
    begin
    --  wait until Connected = true;
        wait until reset = false;
        debug_loop: loop
            spw_get_poll(       -- debug procedure in SpaceWire_Sim_lib
                spw_init_link,  -- variable of protected type containing SpaceWire data buffers
                Dout_p,         -- Data Signal to Debug
                Sout_p,         -- Strobe Signal to Debug
                1               -- Polling timestep (ns)
            );
        end loop debug_loop;
        wait;
    end process;
```

Figure 15: SpaceWire Debug in Testbench

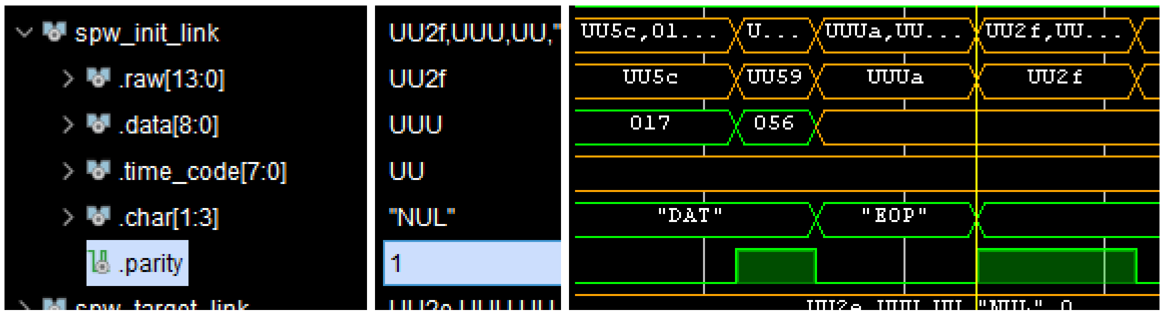


Figure 16: SpaceWire Debug Sim Waveform

Core Example Design (Xilinx Vivado)

This section details bringing up and simulating the IP example design. The testbench consists of a RMAP Initiator IP and an RMAP Target IP both running at 100Mhz. The Initiator sends several read and write requests using both Path and Logical Addressing. For more detail on this, please see the testbench.

Both RMAP Target and RMAP Initiator use the 4Links SpaceWire CoDec IP.

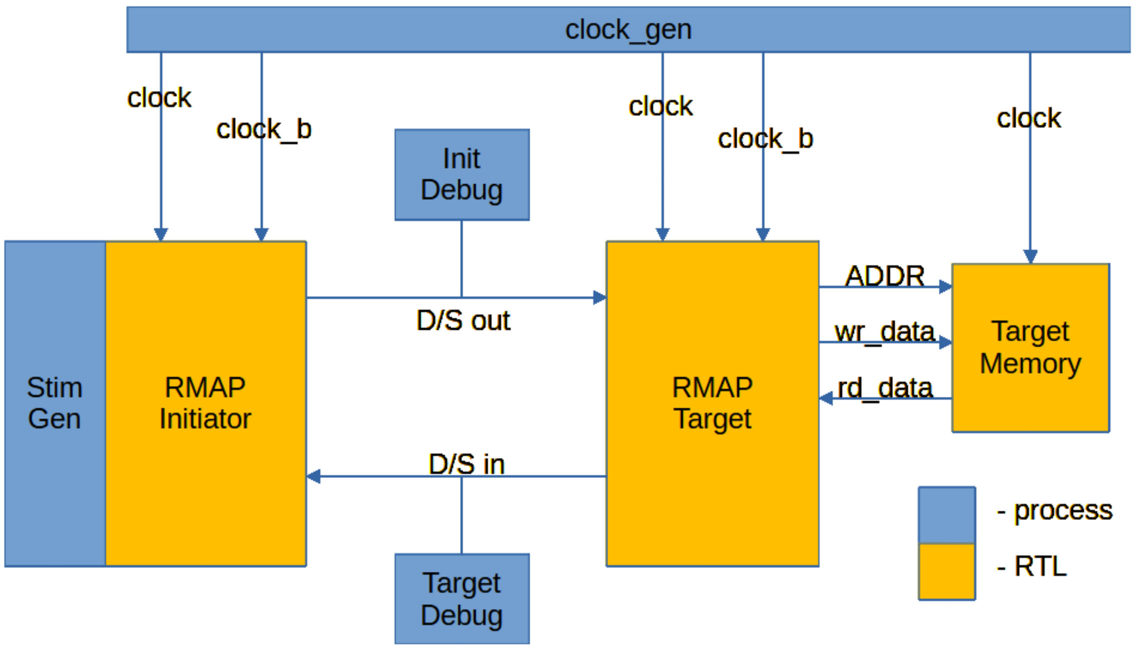


Figure 17: Testbench Architecture

Figure 17 shows the testbench architecture. Clock generation is split between two processes which generate the positive and negative system clocks. All transactions are aligned to the rising-edge of the positive system clock.

All test stimuli are driven through the RMAP Initiator IP. The Debug processes are used to breakout raw SpaceWire D/S signals and check the underlying SpaceWire transactions.

Adding the Project Files

Before copying files into your project, make sure that your VHDL version is set for 2008 or higher.

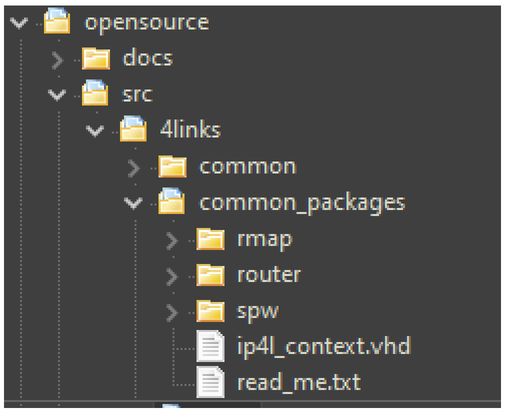


Figure 18: Common Packages Directory

- Import sub-folders from ***common_packages*** into your project. Packages should be added to a library which corresponds to the sub-folder name. i.e files in ***rmap*** should be added to a VHDL library called "rmap".
- Import ***ip4l_context.vhd*** into your project. This creates the context-clauses used for building the IP.

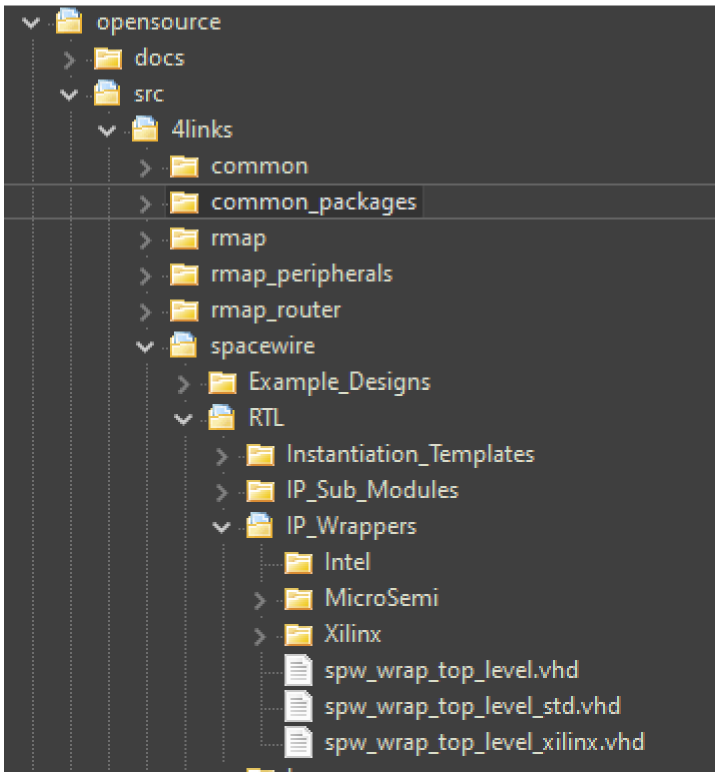
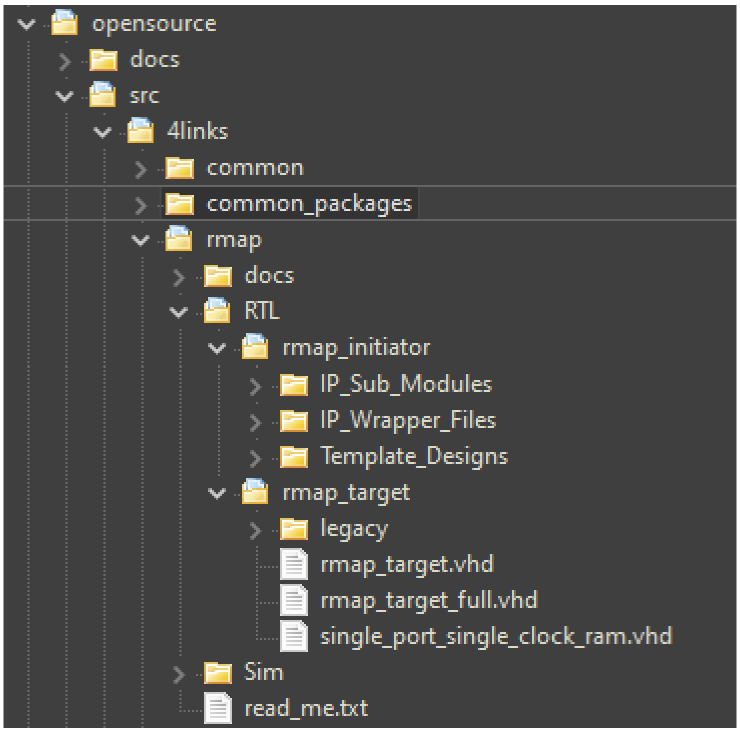


Figure 19: SpaceWire directory

- Import all files from the ***/spacewire/RTL/IP_Sub_Modules*** directory.
- Import ***spw_wrap_top_level.vhd*** and ***spw_wrap_top_level_std.vhd*** from the ***/spacewire/RTL/IP_Wrappers*** directory.



```
opensource
  docs
  src
    4links
      common
      common_packages
      rmap
        docs
        RTL
          rmap_initiator
            IP_Sub_Modules
            IP_Wrapper_Files
            Template_Designs
          rmap_target
            legacy
            rmap_target.vhd
            rmap_target_full.vhd
            single_port_single_clock_ram.vhd
        Sim
        read_me.txt
```

Figure 20: RMAP Directory

- From ***/rmap/RTL/rmap_initiator*** add the ***IP_Sub_Modules*** & ***IP_Wrapper_Files*** directories to your project.
- From ***/rmap/RTL/rmap_target*** add the three .VHD files. Do not add any ***Legacy*** directories to your project.
- Finally, add the ***/rmap/Sim/*** directory to your project. The files here are for testbench only and do not contain synthesizable code.

Setting the ***rmap_initiator_top_level.vhd*** entity as your top-level design entity should allow you to build and synthesize the RMAP initiator. The Initiator interface and SpW options can be configured through use of generics as described in this document.

Setting the ***rmap_initiator_tb.vhd*** as your top-level simulation entity should allow you to run a behavioural simulation of the RMAP Initiator and RMAP Target IP's.

Behavioural Simulation

Once the design has been loaded into your project, you can run a behavioural simulation. This requires that the ***rmap_initiator_top_level.vhd*** has been set as your top-level simulation entity.

The Testbench simulates a large burst write and read to/from the RMAP target memory. Data read back from the target memory is checked against