

# RMAP Router IP User Guide

## Contents

Table of Figures .....	2
0.0 IP Overview .....	3
0.1 Common Terminology .....	3
0.2 IP Files Location.....	4
1.0 IP Architecture .....	7
1.1 Top-Level Configuration.....	7
1.2 IP Sub-Modules .....	8
1.2.1 Port RX Controller .....	8
1.2.2 Routing Table Arbiter .....	9
1.2.3 Port TX Controller .....	9
1.2.4 X-Bar Switch Fabric Top Level .....	9
1.2.5 Port 0 Controller .....	11
1.2.6 Time-Code Logic.....	11
1.2.7 Status & Configuration Registers .....	11
2.0 IP Top-Level Signals.....	12
2.1 Standard Register Control Ports .....	12
2.2 SpaceWire IO Ports .....	12
3.0 Router Configuration (RMAP) .....	13
3.1 Routing Table Configuration .....	15
3.2 Configuration Registers.....	17
3.3 Status Registers.....	17
4.0 Instantiating the IP Core .....	18
5.0 IP Example Design (Xilinx Vivado) .....	19
5.1 Importing the Example Design.....	19
5.2 Simulating the Example Design.....	21
5.3 RMAP Simulation Procedures .....	22
6.0 Omitted Features (Initial Release) .....	23
6.1 SpaceWire Interrupt Registers .....	23
6.2 Port Statistics Registers.....	23
7.0 Arbitration Priority Schemes.....	24
7.1 FiFo Arbitration Methodology .....	24

7.2 FiFo vs Round-Robin Architecture Comparison .....	26
8.0 Resource Utilization (Estimated) .....	27
8.1 12+1 ports (FiFo) .....	27
8.2 12+1 ports (Round Robin) .....	27
8.3 31+1 ports (FiFo) .....	28
8.4 31+1 ports (Round Robin) .....	28

## Table of Figures

Figure 1: 4Links Common Packages directory .....	4
Figure 2: Opensource SpaceWire directory .....	5
Figure 3: Opensource RMAP target directory.....	5
Figure 4: Opensource RMAP Router directory .....	6
Figure 5: IP Sub-Module Architecture (basic) .....	7
Figure 6: router_pckg.vhd configuration Constants .....	7
Figure 7: Top-Level Configuration Constants in router_pckg.vhd .....	8
Figure 8: Toolchain synthesis attributes in dp_fifo_buffer.vhd.....	8
Figure 9: X-Bar Switch Fabric Architecture .....	10
Figure 10: tc_master_mask Constant in router_pckg.vhd.....	11
Figure 11: List of Status/Config Registers in router_pckg.vhd.....	11
Figure 12: r_fifo m/s record signals .....	13
Figure 13: RMAP Address Byte Format.....	14
Figure 14: Router Config/Status Register Modules .....	14
Figure 15: Routing Table Memory Structure .....	15
Figure 16: Example RMAP Write RT Header Bytes .....	16
Figure 17: Example RAMP Write RT Payload Bytes .....	16
Figure 18: Example RMAP Read RT Header Bytes .....	16
Figure 19: SpaceWire Configuration Registers Example.....	17
Figure 20: 4Links Common Packages Files location .....	19
Figure 21: RMAP Target IP Files Directory .....	19
Figure 22: RMAP Router RTL Files Directory.....	20
Figure 23: Example Design Testbench location .....	20
Figure 24: req_active signal waveform.....	21
Figure 25: Simulation; SpW Debug Signals .....	22
Figure 26: rmap_frame testbench Example .....	23
Figure 27:FiFo Arbitration: Checking Process .....	24
Figure 28: FiFo Arbitration: Shifting Process.....	25
Figure 29: FiFo based arbitration results .....	26
Figure 30: Fair round-robin arbitration results.....	26

## 0.0 IP Overview

The 4Links Ltd RMAP Router IP is an open-source SpaceWire Router IP with support for up to 32 routing ports (1 virtual, 31 physical). The IP Core is written entirely in VHDL and is left unencrypted to allow for modification by the end user. An example design is provided with the IP which this document will cover. The IP Example design is targeting Xilinx Vivado toolchain and KCU116 FPGA Development board (Kintex UltraScale+).

### 0.1 Common Terminology

- **SpaceWire Source** – a source of SpaceWire data in a network, usually a RMAP Initiator.
- **SpaceWire Sink** – destination of SpaceWire data in a network, such as a RMAP Target.
- **X-Bar Switch** – a **Crossbar** switch. Allows for non-blocking communication between multiple Inputs and Outputs.
- **X-Bar Switch Fabric** – The internal interconnects between IO on a Crossbar switch.
- **Byte** – 8 Bits
- **Word** – 16 Bits
- **D-Word** – Double Word, 32 Bits.
- **t\_ports** – a std\_logic array with length N, where N is the total number of ports on router.
- **Slv** - Standard Logic Vector, VHDL data type.

When referring to the number of SpaceWire ports on a router, the total number of ports is determined by the number of physical routing ports + one virtual port (port 0). Therefore a 13-port router would have 12 physical ports and 1 virtual port (12+1). The maximum number of configurable ports via the **router\_pckg.vhd** configuration file is 32, which allows for 31 physical ports and 1 internal port (31+1).

## 0.2 IP Files Location

The router IP files are located under the **rmap\_router** sub-directory. Before adding the rmap router to your project, you must first add its dependencies. This section will cover how to add these to your project.

1. Add **common\_packages** to your project.

The **common\_packages** sub-directory contains the necessary VHDL library and package files to build 4Links opensource IP.

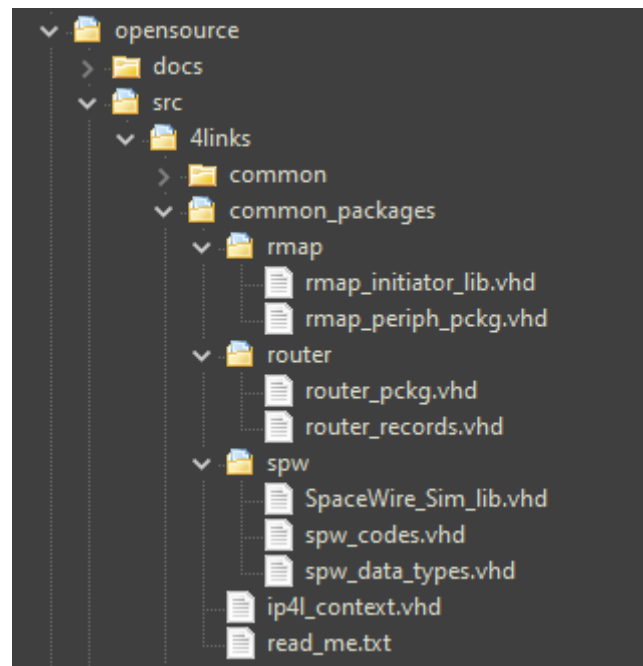


Figure 1: 4Links Common Packages directory

There are three sub-directories, rmap, router and spw. The files within each directory should be added to a VHDL library. The name of each library is the sub-folder name. For example, **SpaceWire\_Sim\_lib.vhd** should be added to a VHDL library names **spw**. The **ip4l\_context.vhd** file should be added to the current working directory (.work library).

## 2. Add the SpaceWire CoDec IP Files

The SpaceWire CoDec and its wrapper files are used to build the RMAP Router IP.

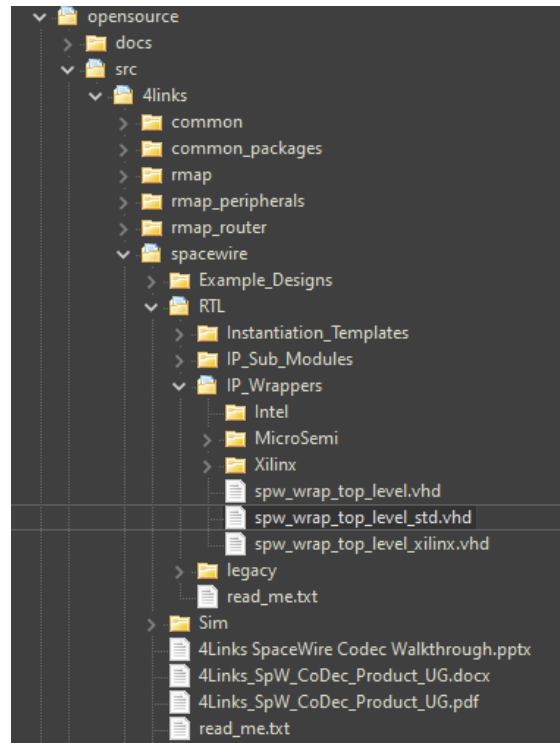


Figure 2: Opensource SpaceWire directory

All files from the **IP\_Sub\_Modules** folder should be added to your project. Wrapper file **spw\_wrap\_top\_level.vhd** should be added to your project. This wrapper converts the legacy boolean type interfaces on the SpaceWire codec to std\_logic types.

## 3. Add the required RMAP Target source files

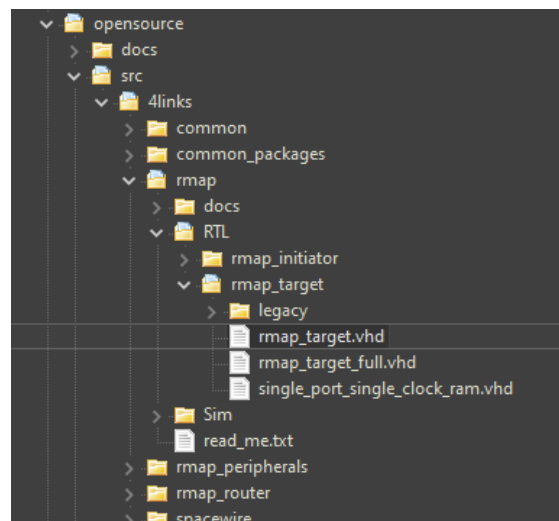


Figure 3: Opensource RMAP target directory

From the **rmap/RTL/rmap\_target** directory, add the **rmap\_target.vhd** IP file. The RMAP target is used for the virtual port (port 0) controller.

4. Finally, Add the router IP files and Example testbench

Add all files from the **IP\_Sub\_Modules** directory, excluding any **legacy** folders. Add the **router\_top\_level.vhd** file to your design. Add **router\_top\_level\_tb.vhd** to your project as a simulation source.

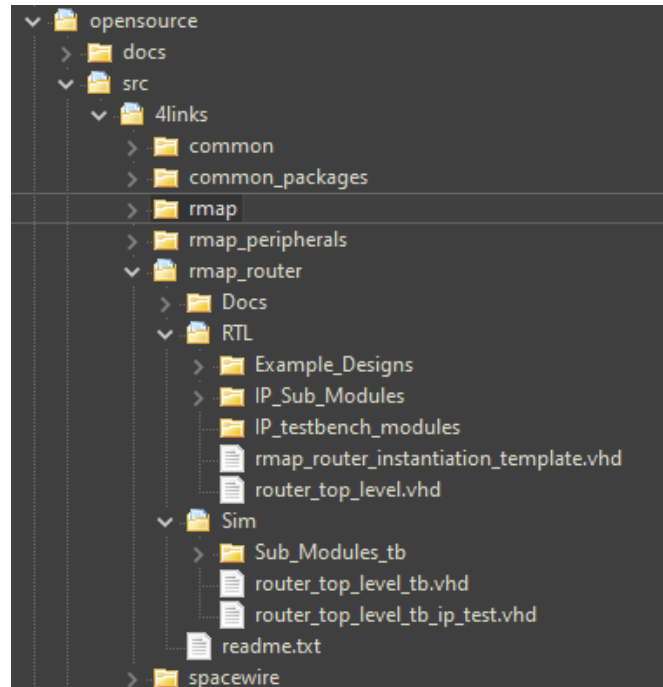


Figure 4: Opensource RMAP Router directory

Setting the **router\_top\_level.vhd** file as your top-level design entity, the router should now elaborate and build. A behavioural simulation can be run using the provided top-level testbench.

## 1.0 IP Architecture

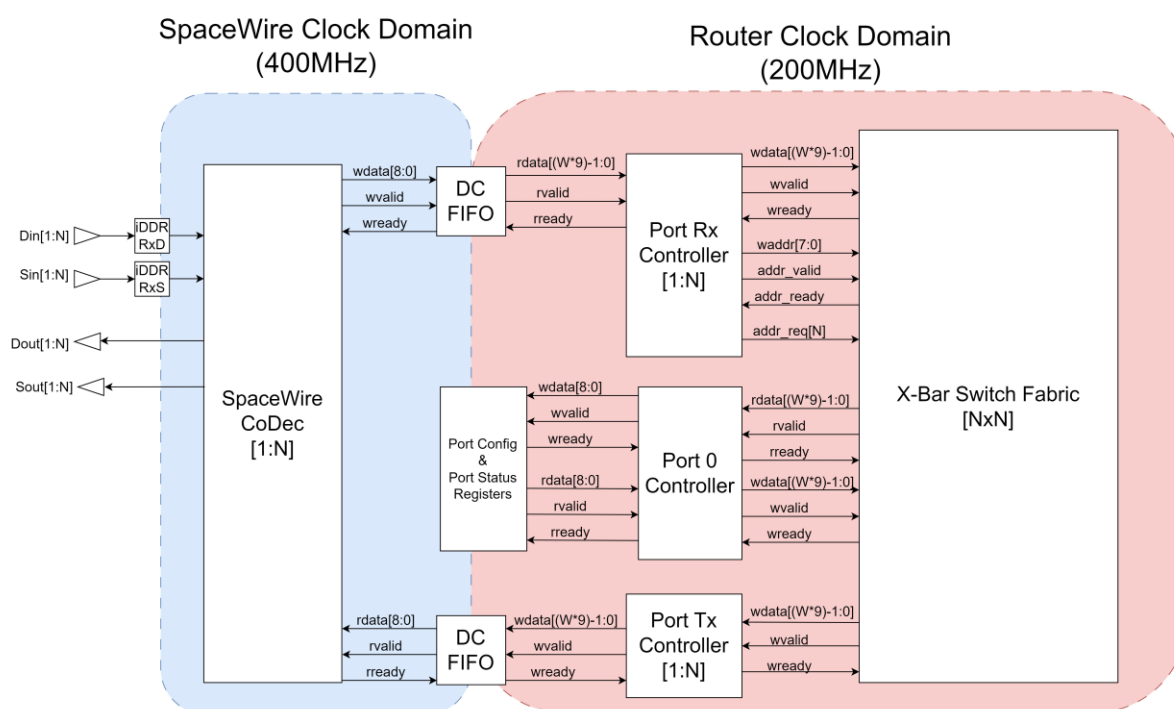


Figure 5: IP Sub-Module Architecture (basic)

### 1.1 Top-Level Configuration

The router can be configured using top-level generics as well as package-based constants. For changing the number of generated ports, it is required that you modify the constant **c\_num\_ports** located on line 38 in **router\_pkg.vhd**. Simply changing the top-level generic in **router\_top\_level.vhd** can work but could lead to issues in synthesis as several types make use of the **c\_num\_ports** constant.

NAME	TYPE	DEFAULT	BRIEF
c_fabric_bus_width	natural range 1 to 4	2	nonet-width of Xbar fabric
c_spw_clk_freq	real	400_000_000.0	SpaceWire Port Clock Frequency
c_router_clk_freq	real	200_000_000.0	Routing Fabric Clock Frequency
c_num_ports	natural range 1 to 32	13	Total number of ports on design
c_port_mode	string	"custom"	Spw port IO mode (keep custom)
c_priority	string	"fifo"	Router Fabric arbitration scheme
c_ram_style	string	"block"	Synthesis Tool ram attribute
c_tc_master_mask	t_dword	--	Set TimeCode master port
c_fifo_ports	t_dword	(others => '0')	Set SpW port as FIFO port, no Codec

Figure 6: router\_pkg.vhd configuration Constants

```

-- Global config constants --
-- router top level generic configuration constants --
constant c_fabric_bus_width      : natural range 1 to 4      := 2;          -- nonet-wi
constant c_spw_clk_freq          : real                      := 400_000_000.0; -- frequenc
constant c_router_clk_freq       : real                      := 200_000_000.0; -- frequenc
constant c_num_ports             : natural range 2 to 32      := 13;         -- number o
constant c_port_mode             : string                   := "custom";   -- valid op
constant c_priority              : string                   := "fifo";     -- valid op
constant c_ram_style             : string                   := "block";    -- type of
constant c_tc_master_mask        : t_dword                 := b"0000_0000_0000_0000_0000_0010_0000";
constant c_fifo_ports            : t_dword                 :=( -- select which ports are fifos. Ports outs
    0 => '0',
    1 => '1',
    -- 0 has no effect, always '0' as Port 0 is an internal port

```

Figure 7: Top-Level Configuration Constants in router\_pckg.vhd

The constant **c\_fabric\_bus\_width** can be used to modify the bus width for the crossbar fabric. More detail on this will be provided in later sections of this guide. For reference, increasing this value allows for a lower router\_clk frequency (to meet throughput targets) at the expense of more routing resources.

```

134
135 -- Attribute Declarations --
136
137 attribute ram_style      : string; -- Vivado
138 attribute ramstyle       : string; -- Quartus
139 attribute syn_ramstyle   : string; -- Libero
140
141 attribute ram_style      of s_ram : variable is g_ram_style; -- declare ram style (Vivado synthesis attribute)
142 attribute ramstyle       of s_ram : variable is g_ram_style; -- declare ram style (Quartus synthesis attribute)
143 attribute syn_ramstyle   of s_ram : variable is g_ram_style; -- declare ram style (Libero synthesis attribute)
144

```

Figure 8: Toolchain synthesis attributes in dp\_fifo\_buffer.vhd

The ram\_style constant is passed to the attributes (Figure 8) located in the **dp\_fifo\_buffer.vhd** file. The file contains attributes for ram\_style for several popular device toolchains. Vivado will ignore invalid synthesis attributes so the attributes for Quartus and Libero can be left as is. Your millage may vary when using Quatus or Libero, so it is advised that you check your toolchain documentation and adjust as required.

## 1.2 IP Sub-Modules

As detailed in Figure 5, the IP is constructed from several sub-modules. This section is dedicated to describing the function and operation of each sub-module.

### 1.2.1 Port RX Controller

Reads bytes from the connected RMAP Data source (Codec/RMAP Target for Port 0) and checks for correct format. First valid byte is passed to the Routing table. The Rx Controller will assert a request line to the X-Bar Switch. This notifies the X-bar switch that this port is requesting target access. Once the routing table has processed the request and the X-Bar fabric connected, an axi-style data channel is opened between the requester port (Rx Controller) and the Target ports (Tx Controller(s)).

The RX Controller will strip path addresses from the RMAP data stream if required. Logical addresses are automatically pushed to the TX Interface once connection has been established across the X-Bar fabric.

To prevent time-outs and lockups, any disconnected Target ports should hold their **spw\_data\_ready** ports high. This will allow data to be discarded without locking up the router. The 4Links SpaceWire IP performs this automatically, however, user controlled FiFo ports should take care to implement this behaviour.



Earlier versions of this IP core included an in-built, user-configurable time-out for transactions across the router. This was removed in favour of end-user flexibility. Time-outs are expected to be handled in the user application, rather than within the Router IP.

### 1.2.2 Routing Table Arbiter

The Routing Table Arbiter is responsible for implementing a fair round-robin arbitration scheme for routing table access between the (up to) 32 SpaceWire ports. Arbitration is required to ensure that routing table access is concurrently safe.

The arbiter uses a two-stage masked-grant implementation with ascending priority. The priority pointer is rotated with each transaction. If the priority port is not active, then the next highest priority port will be selected and granted access to the routing table. The implementation is entirely fair to prevent port starvation. This ensures that no matter the number of requests, for N number of ports, the request will be queued for a maximum of N-1 request transactions.

The Routing Table is stored as a 256x32Bit (1024x8bit) Mixed-Width RAM within the arbiter architecture.

### 1.2.3 Port TX Controller

The TX controller asserts its data line ready once it receives a valid signal from a Port RX Controller. Since the RX Controller is responsible for sending X-Bar switch configuration settings, the TX Controller simply needs to pass data it receives to the connected SpaceWire data sink (SpaceWire Port).

The TX Controller is not required to detect EEPs/EOPs and simply acts as an interface into the SpaceWire sink.

### 1.2.4 X-Bar Switch Fabric Top Level

The Crossbar Switch Fabric is responsible for connecting the 32x SpaceWire Requesters/Targets whilst allowing for concurrent data transactions between ports. A non-blocking crossbar switch architecture is implemented using a combination of 1-32 and 32-1 Multiplexers. Target/Requester port verification is implemented to prevent port conflicts in the event of an arbitration failure.

The switch fabric allows for true-multicasting and concurrent communication between Rx and Tx controller ports. Port 0 on the fabric is connected to the virtual SpaceWire Port 0 which contains an RMAP target.

To improve timing performance, pipeline registers are inserted at several stages within the fabric architecture. The registers have no associated resets to improve timing, therefore when performing a system reset, the reset should be held for 5 clock cycles.

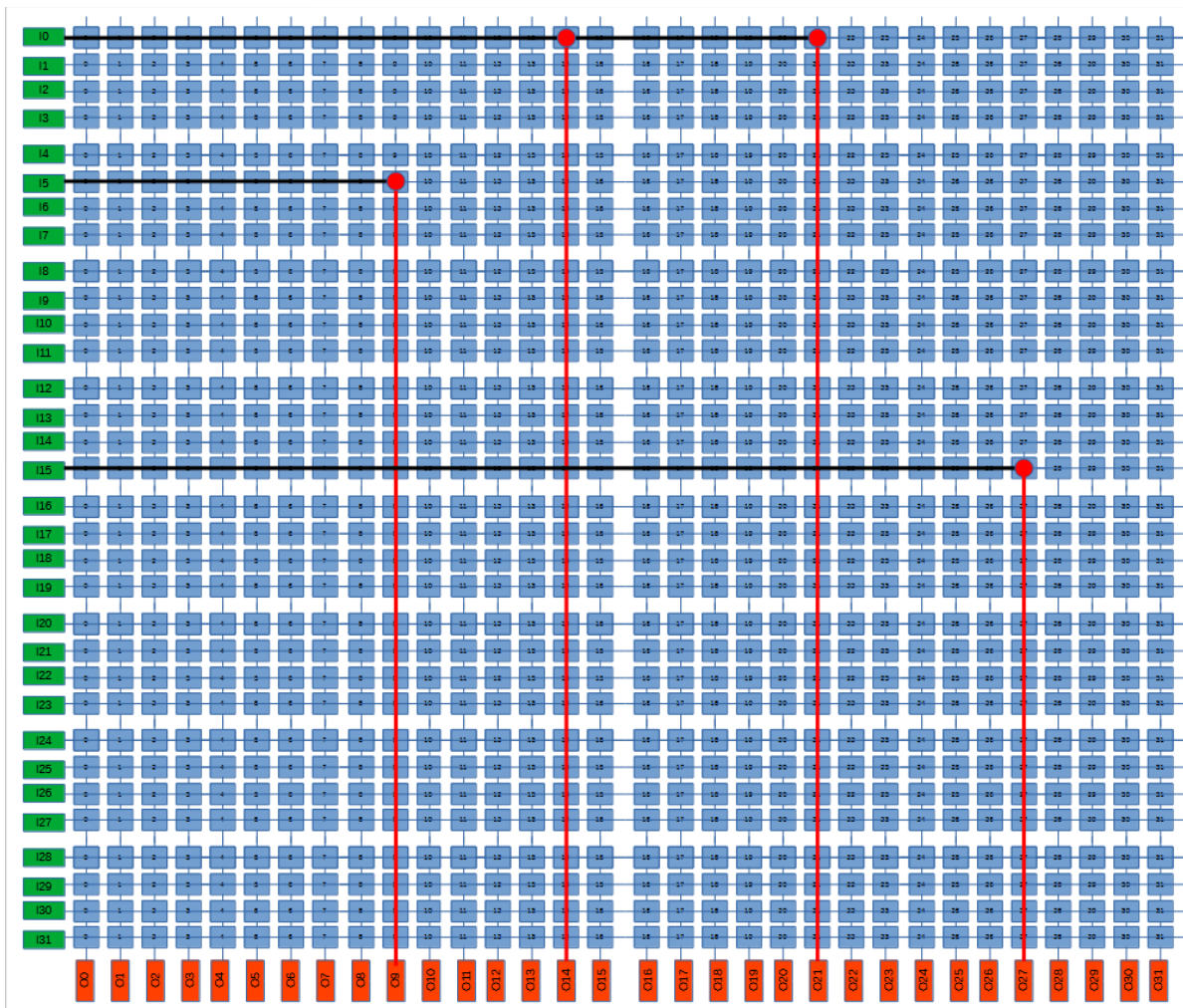


Figure 9: X-Bar Switch Fabric Architecture

Figure 9 shows how the x-bar switch fabric is arranged. Where each node on the switch joins an output to an input. Multiple Outputs (targets) can be selected for a single Input (requester). The opposite however is not true. For a 32x32 Switch 1024 “nodes” are required. Henceforth the X-Bar Fabric and its control make up most the IP’s logical resources.

In scenarios where multi-cast is enabled, where a single requester connects to multiple targets, the requester-ready signal is only asserted when all the target ports are ready. Therefore, if one of the targets is queued for arbitration, the requester transaction will be held until all ports are available. Care must be taken as one (slow) queued port can delay the entire multicast transaction. Therefore, it is a better strategy to perform many small (burst) transactions rather than one large continuous transaction in scenarios where lots of data is to be transferred between RMAP devices.

When configuring the number of ports on the router, the dimensions of the X-Bar switch fabric is automatically trimmed down from 32x32 to NxN, where N is the total number of ports on the router. This significantly reduces resource usage when instantiating router IP with low numbers of ports.

The internal function and implementation of the x-bar fabric and its arbitration methods are outside the scope of this document. A detailed comparison between the FiFo and round-robin fabric arbitration methods can be found in the 7.0 Arbitration Priority Schemes section of this document.

### 1.2.5 Port 0 Controller

This acts as a virtual SpaceWire port connected to an RMAP Target. This allows for internal configuration and monitoring of the RMAP Router. All ports can access this target by using the path address 0x00 in their address byte submission.

The RMAP Key is set within the ***router\_pckg.vhd***. Writing to the RMAP target should write data to the internal configuration registers. Reading from the target will read from the router status registers.

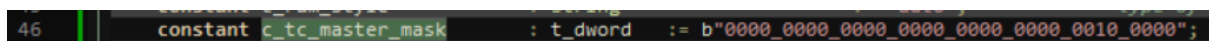
Importantly, Port 0 can be used to read/write the Routing Table address spaces. On system start-up, connected RMAP devices can write address data to the routing table and then use RMAP read replies to broadcast this information to connected devices on the network.

### 1.2.6 Time-Code Logic

The timecode logic is responsible for checking and broadcasting submitted timecodes to all connected devices on the network. The value of the last received timecode can be read from Port 0 using the relevant status registers. The address of the TimeCode status register can be configured in the ***router\_pckg.vhd*** package file. By default, this register has address 0x03, 0x00. Where 0x03 is the module address and 0x00 is the timecode register address within the module.

Selecting the Timecode Master is done using the ***misc\_config\_register*** at address 0x04. The 32-Bit mask is stored across registers 0x00 to 0x03. To select the time-code master, set one of these bits high. Alternatively, the Timecode master can be set at synthesis using the ***c\_tc\_master\_mask*** constant in ***router\_pckg.vhd***. Setting a single bit high will set that port at the time-code master mask.

Only a single time-code mask bit should be set high, setting multiple high can result in the time-code broadcast logic locking-up.



```
46      constant c_tc_master_mask : t_dword := b"0000_0000_0000_0000_0000_0000_0010_0000";
```

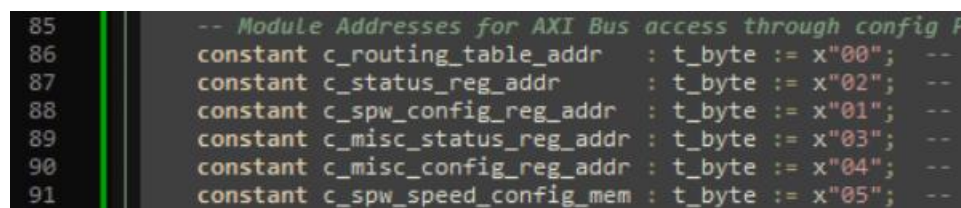
Figure 10: *tc\_master\_mask* Constant in *router\_pckg.vhd*

Non-connected SpaceWire Ports are automatically masked off to prevent hang-ups when broadcasting a new time code to Time-code slave ports.

### 1.2.7 Status & Configuration Registers

These registers support both RMAP write and RMAP read commands. They are responsible for configuring the router and its' ports. Unlike configuration registers, status registers are read only.

Status and configuration registers are split into addressable modules, each module contains 32 addressable, byte-sized, register spaces.



```
85      -- Module Addresses for AXI Bus access through config P
86      constant c_routing_table_addr : t_byte := x"00"; --
87      constant c_status_reg_addr   : t_byte := x"02"; --
88      constant c_spw_config_reg_addr : t_byte := x"01"; --
89      constant c_misc_status_reg_addr : t_byte := x"03"; --
90      constant c_misc_config_reg_addr : t_byte := x"04"; --
91      constant c_spw_speed_config_mem : t_byte := x"05"; --
```

Figure 11: List of Status/Config Registers in *router\_pckg.vhd*

## 2.0 IP Top-Level Signals

Not all top-level signals will be connected in your elaborated design. Connections will depend on the on the configuration generics and constants within the design. For a full guide on how these constants can be modified see 4.0 Instantiating the IP Core.

Each SpaceWire IO signal is represented a `std_logic_vector`. Where the number of elements in the `slv` represents the number of physical ports on the router. To conform with port numbering conventions, ports start from values (1) to N where  $N < 32$  and the number of configured ports on the design.

### 2.1 Standard Register Control Ports

NAME	TYPE	STATUS	BRIEF
spw_clk_p	std_logic	Active high	SpW Port Clock Input (P)
spw_clk_n	std_logic	Active high	SpW Port Clock Input (N)
router_clk	std_logic	Active High	Router Clock Input
rst_in	std_logic	Active high	Active-high synchronous Reset
enable	std_logic	Active high	Active-high synchronous Enable

### 2.2 SpaceWire IO Ports

When using ***g\_mode = "custom"*** the following SpaceWire IO ports are used:

NAME	TYPE	STATUS	BRIEF
DDR_din_r	slv(1 to N)	Sampled by clk	Rising edge sample of D-in Port(s)
DDR_din_f	slv(1 to N)	Sampled by clk_b	Falling edge sample of D-in Port(s)
DDR_sin_r	slv(1 to N)	Sampled by clk	Rising edge sample of S-in Port(s)
DDR_sin_f	slv(1 to N)	Sampled by clk_b	Falling edge sample of S-in Port(s)
SDR_Dout	slv(1 to N)	Updated on clk	SpW Port Data Output
SDR_Sout	slv(1 to N)	Updated on clk	SpW Port Strobe Output

If using ***g\_mode = "single" / "diff"*** then the following alternative IO is used:

NAME	TYPE	STATUS	BRIEF
Din_p	slv(1 to N)	LVDS Din Positive	Used as Single Ended Input when "single"
Din_n	slv(1 to N)	LVDS Din Negative	
Sin_p	slv(1 to N)	LVDS Sin Positive	Used as Single Ended Input when "single"
Sin_n	slv(1 to N)	LVDS Sin Negative	
Dout_p	slv(1 to N)	LVDS Dout Positive	Used as Single Ended output when "single"
Dout_n	slv(1 to N)	LVDS Dout Negative	
Sout_p	slv(1 to N)	LVDS Sout Positive	Used as Single Ended output when "single"
Sout_n	slv(1 to N)	LVDS Sout Negative	

**Type *t\_ports* is a `std_logic` array with length number of SpaceWire ports. Note that**

Note that the use of ***"single" or "diff"*** modes is recommended for simulation only. For synthesis, you will need to implement the relevant DDR registers and IO buffers as covered in the 4Links SpaceWire Codec user guide.

Alternatively, if a port is configured to use a FIFO interface, rather than a SpaceWire Codec, neither of these interface options will be used. Instead, the port will be connected to the relevant *spw\_fifo* interface.

NAME	TYPE	STATUS	BRIEF
spw_fifo_in	Array of r_fifo_master	N/A	Used when port is configured as FIFO port
spw_fifo_out	Array of r_fifo_slave	N/A	Used when port is configured as FIFO port

The signals and types belonging to each of the interfaces can be seen in Figure 12.

```

77  type r_fifo_master is record
78      rx_data      : t_nonet;
79      rx_valid     : std_logic;
80      rx_time      : t_byte;
81      rx_time_valid : std_logic;
82      tx_ready     : std_logic;
83      tx_time_ready : std_logic;
84      connected    : std_logic;
85  end record r_fifo_master;
86
87  type r_fifo_slave is record
88      tx_data      : t_nonet;
89      tx_valid     : std_logic;
90      tx_time      : t_byte;
91      tx_time_valid : std_logic;
92      rx_ready     : std_logic;
93      rx_time_ready : std_logic;
94  end record r_fifo_slave;

```

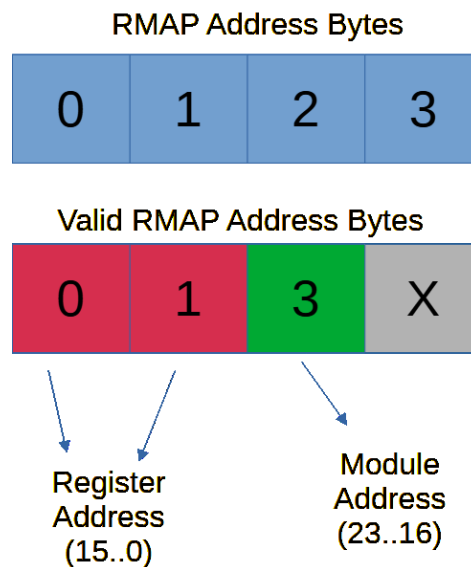
Figure 12: r\_fifo m/s record signals

For instantiation into a non-VHDL design source, it is recommended that the top-level design file is first wrapped to expose the internal record ports. Some toolchains struggle with record-extraction when moving between VHDL and System Verilog/Verilog.

The FIFO master/slave interfaces give direct access to the Data and Timecode channels for the router. Information is exchanged using an axi-like ready/valid handshake. Where a transaction is valid only when both ready and valid signals are asserted.

### 3.0 Router Configuration (RMAP)

Configuring and monitoring the router is achieved by using the virtual port (address 0x00). The port implements a full RMAP target to process incoming commands. The RMAP Target Read/Write interface is connected to the router Status and Configuration registers, as well as the Routing Look-up Table. Of the nominal 32-bit RMAP address, only the lower 3 bytes are used. Bits (23 down to 16) is the internal module address (Status reg /Config reg /Routing Table). The lower bits (15 down to 0) are used to address registers/memory elements within the module (see Figure 13). The address for each module can be configured in the package file *router\_pkg.vhd*.



*Figure 13: RMAP Address Byte Format*

Status and Configuration register space can be modified as required to include required information about the router. It is recommended that the “misc” status and configuration register spaces are used for this as they contain unused bytes.

<u>Module Address</u>	<u>Module Name</u>	<u>Address Space Size Length x Width</u>
0x00	Routing Table	1024x8
0x01	SpW Port Config Registers	32x8
0x02	SpW Port Status Registers	32x8
0x03	Misc Status Registers	32x8
0x04	Misc Config Registers	32x8
0x05	Port pre-scalar Registers	32x8

*Figure 14: Router Config/Status Register Modules*

### 3.1 Routing Table Configuration

The routing address map is stored as a 256x32-bit memory element. Where the 32-bit data D-Word uses one-hot encoding to store the target address port map. The address range 0-255 corresponds to the RMAP address byte value. All incoming request data bytes are posted to the routing table to determine the target output port(s).

To make reading/writing configuration data easier using RMAP. The routing look-up table has mixed width ports. The write interface to the look-up table uses a 1-byte wide interface, to match the RMAP interface. The Read interface is 32-bits as required by the routing logic. When writing a new value to a table address, you must write 4 bytes to the address pointer. The address pointer is 4x the address number. See Figure 15 for an example of this behaviour. Numbers on the left-side denote the read addressed, where numbers on the right side denote write addresses.

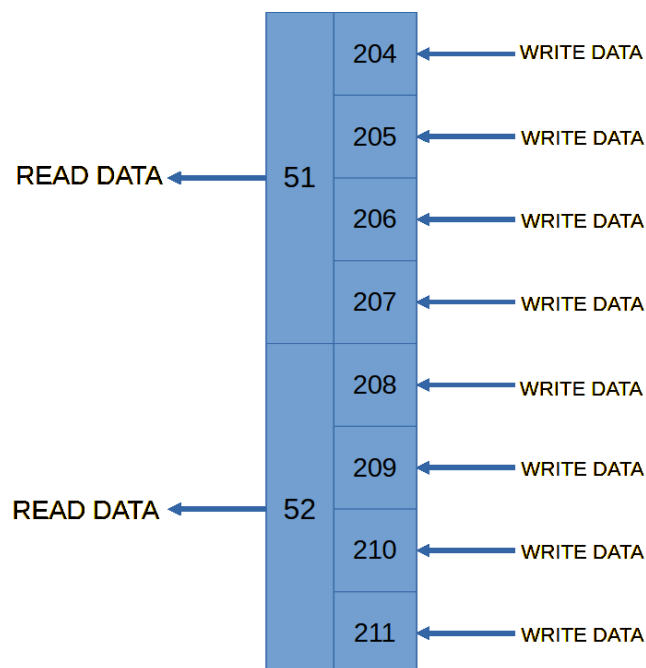


Figure 15: Routing Table Memory Structure

For example:

We want to configure the RMAP router so that an RMAP packet sent with logical address byte 0x33 is multicast routed to output ports 21 and 14.

To update the routing table address for 0x33 (51 in decimal) we write 4 bytes of data to the following address spaces at port 0.

- 0x0000\_00CC, where 0xCC (204) is 4x 0x33 (51).
- 0x0000\_00CD (205)
- 0x0000\_00CE (206)
- 0x0000\_00CF (207)

The first byte written (0xCC) is the LS-Byte of the 32-bit D-Word address and the last byte written (0xCF) is the MS-Byte of the 32-Bit D-Word address. For the target ports 21 and 14, bits (21) and (14) are expected to be high whilst others are low. Therefore, the Bytes would look like:

- B0 0b0000\_0000 (0x00)



- B1 0b0100\_0000 (0x40)
- B2 0b0010\_0000 (0x20)
- B3 0b0000\_0000 (0x00)

The RMAP target at Port 0 supports increment Read/Writes. Therefore, this process can be simplified by simply sending an RMAP increment write instruction with the desired 4-Byte payload to the starting address pointer at the routing table (in this case 0xCC).

A shadow memory element is implemented with the routing table memory. The shadow memory allows for RMAP reads to be performed on the routing-table memory without interrupting port access. The Shadow memory uses a true 1024x8bit RAM which allows for RMAP Increment Read operations. The initial address for an RMAP read operation (using address increment) should be set to the address pointer. 4 bytes should be read to get the 32-bits of table data stored at the address.

Figure 16 through Figure 18 show the RMAP header & payload structures required for the RMAP Write/Read operations detailed above.

```
constant c_router_header_pattern_1 : t_byte_array(0 to 15) :=(
  0 => x"00", -- path address
  1 => x"FE", -- target logical address
  2 => x"01", -- protocol ID
  3 => b"0110_0100", -- Instruction
  4 => x"01", -- key
  5 => x"33", -- initiator logical address
  6 => x"00", -- transaction ID MSB
  7 => x"00", -- transaction ID LSB
  8 => x"00", -- extended address
  9 => x"00", -- address MSB(3)
  10 => x"00", -- address(2)
  11 => x"00", -- address(1)
  12 => x"CC", -- address LSB (0)
  13 => x"00", -- data length MSB(2)
  14 => x"00", -- data length (1)
  15 => x"04" -- data length LSB(0)
);
```

Figure 16: Example RMAP Write RT Header Bytes

```
constant c_router_data_pattern_1 : t_byte_array(0 to 3) :=(
  0 => b"0000_0000", -- LSB
  1 => b"0100_0000",
  2 => b"0010_0000",
  3 => b"0000_0000" -- MSB
);
```

Figure 17: Example RAMP Write RT Payload Bytes

```
constant c_router_header_pattern_3 : t_byte_array(0 to 15) :=(
  0 => x"00", -- path address
  1 => x"FE", -- target logical address
  2 => x"01", -- protocol ID
  3 => b"0100_1100", -- Instruction
  4 => x"01", -- key
  5 => x"05", -- initiator logical address
  6 => x"00", -- transaction ID MSB
  7 => x"00", -- transaction ID LSB
  8 => x"00", -- extended address
  9 => x"00", -- address MSB(3)
  10 => x"00", -- address(2)
  11 => x"00", -- address(1)
  12 => x"CC", -- address LSB (0)
  13 => x"00", -- data length MSB(2)
  14 => x"00", -- data length (1)
  15 => x"04" -- data length LSB(0)
);
```

Figure 18: Example RMAP Read RT Header Bytes



### 3.2 Configuration Registers

Configuration registers are divided into 32-byte sections, each section using a different module address. The following are a list of configuration register spaces and their addresses:

- 0x01 – SpaceWire CoDec Configuration Registers
- 0x04 – Misc Configuration Registers

Each byte in the SpaceWire (0x01) register space (0x00 – 0x1F) corresponds to a possible SpaceWire Port. (0 to 31). If less than 32 ports are used, then reading/writing to the unused register space will have no effect.

Register locations remain constant, regardless of the number of generated SpaceWire ports.

Module addresses can be configured in the ***router\_pckg.vhd*** package file.

The Misc configuration register contains other router configuration bytes. This also includes the one-hot-encoded Time-Code Master Mask-bytes. These are in bytes 0x00 to 0x03 of the register space. The module address for this register space is 0x04.

The Bit mapping for each SpaceWire CoDec Configuration Register is as follows:

- 0 – Disable, assert to disable the SpaceWire CoDec
- 1 – Error Inject, assert to Inject Error Code
- 2..5 – Error Select, assert bits to select Error Code to inject
- 6..7 – RESERVED

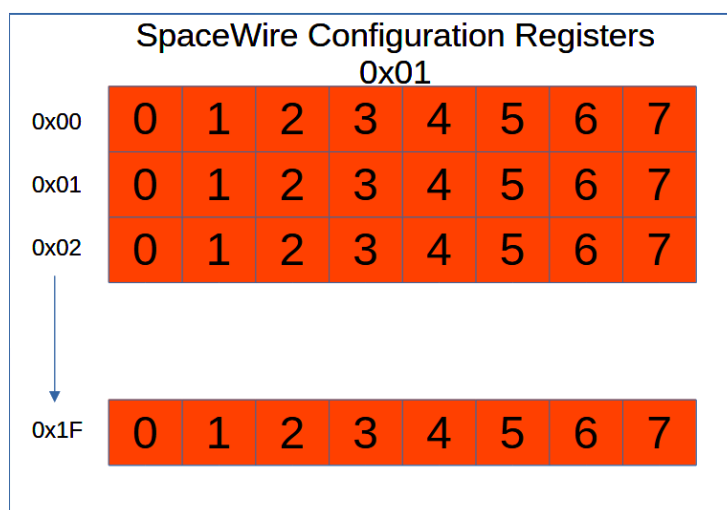


Figure 19: SpaceWire Configuration Registers Example

### 3.3 Status Registers

Like configuration registers, status registers are divided into 32-byte sections. The following are a list of sections and their register addresses:

- 0x02 – SpaceWire CoDec Status Registers
- 0x03 – Misc Status Registers.

The Misc status registers can be used to read generic router status information, this includes the last time-code byte (address 0x20).

The Bit Mapping for Each SpaceWire CoDec Status Register is as follows:

- 0 – Connected, High when SpW Link is active
- 1 – RX\_ESC\_ESC, High when double-ESC character ERROR
- 2 – RX\_ESC\_EOP, High when ESC-EOP detected
- 3 – RX\_ESC\_EEP, high when ESC-EOP detected
- 4 – RX Parity Error, high when SpaceWire Parity Error on RX data
- 5..7 – RESERVED

## 4.0 Instantiating the IP Core

To instantiate the IP core into your design you can use the instantiation template found in ***rmap\_router\_instantiation\_template.vhd***. You may need to create an additional wrapper for your design if instantiating within a non-VHDL project. The record interfaces can be found within ***router\_records.vhd*** package file.

IO can be left disconnected if not required. Default IO states for record interface types can be found in ***router\_records.vhd*** package.

## 5.0 IP Example Design (Xilinx Vivado)

The Router IP comes with an Example design targeting the Xilinx XCKU040 FPGA. The design is for a 13 Port (12+1) Router Implementation @ 400MHz SpaceWire Line-rate. All Ports in the example design use a SpaceWire CoDec, no fifo ports are instantiated.

### 5.1 Importing the Example Design

In the absence of any tcl scripts, the following steps can be taken to import the RMAP Router and Example design into your project.

First, set your project VHDL version to 2008. The Example design and Router IP files rely on constructs which are only present in VHDL 2008 and onwards.

- Import all common package files from the 4Links Opensource Directory.

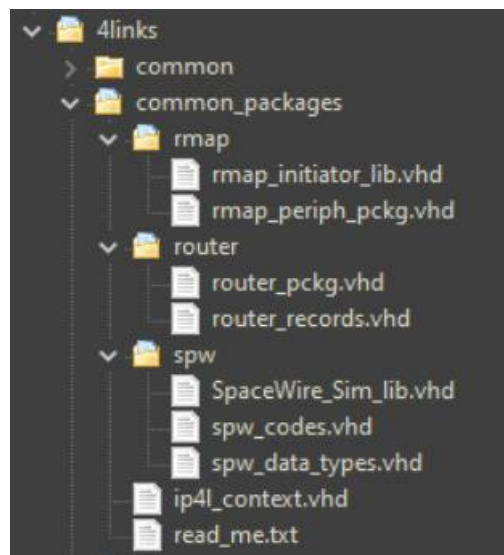


Figure 20: 4Links Common Packages Files location

- Import the 4Links SpaceWire CoDec RTL files and top-level wrappers (see SpaceWire Codec user guide)
- Import all the RMAP Target IP RTL files (Excluding Legacy Subfolder)

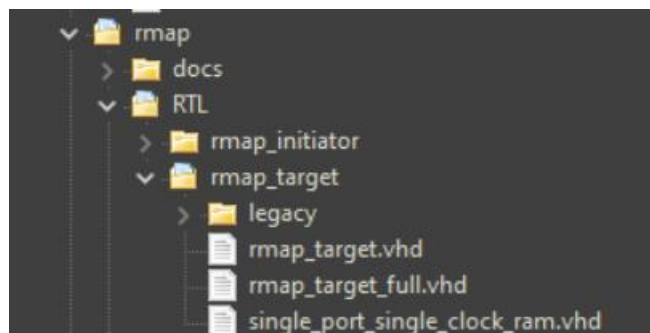


Figure 21: RMAP Target IP Files Directory

- Import the RMAP Router RTL files (excluding Legacy sub-folder and Instantiation Template)

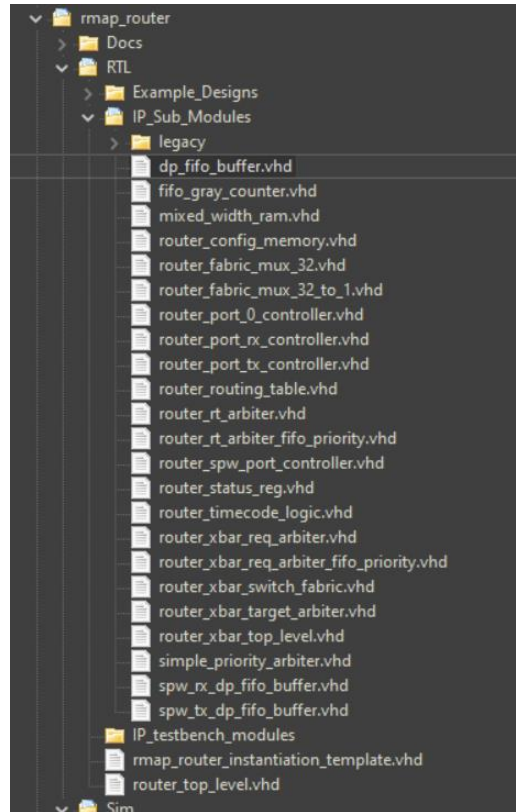


Figure 22: RMAP Router RTL Files Directory

- Import the Router Testbench as a simulation file

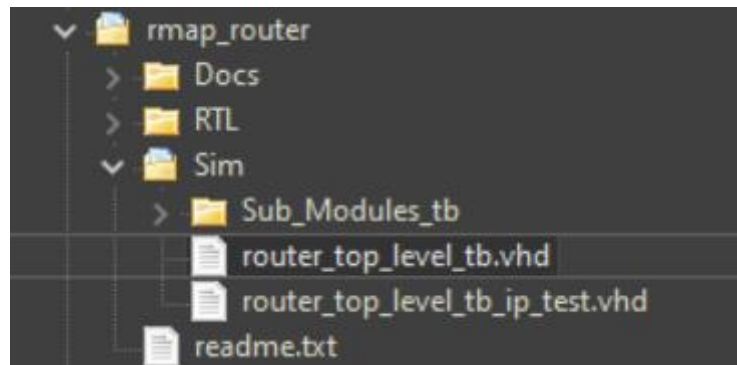


Figure 23: Example Design Testbench location

If using the ip\_test testbench file, then import the RMAP initiator files as well. If not, importing the RMAP Initiator can be ignored.

## 5.2 Simulating the Example Design

The testbench included with the example design is automatically configured when setting the configuration constants in **router\_pckg.vhd**. In this testbench, concurrent RMAP transactions occur across each port. Each port also performs an RMAP Write/Read to/from the routing table.

After writing to the routing table, each port then performs a multicast request for the same set of ports. This is intended to stress test the X-Bar fabric and port arbitration schemes. Once all stimuli have finished, the testbench will stop. If, for some reason, the stimuli cannot be finished, the testbench will time-out after simulating 2500.0 us.

Number of ports and design clock frequency is set in **router\_pckg.vhd**. A SpaceWire debug signal is attached to the output of each SpaceWire port.

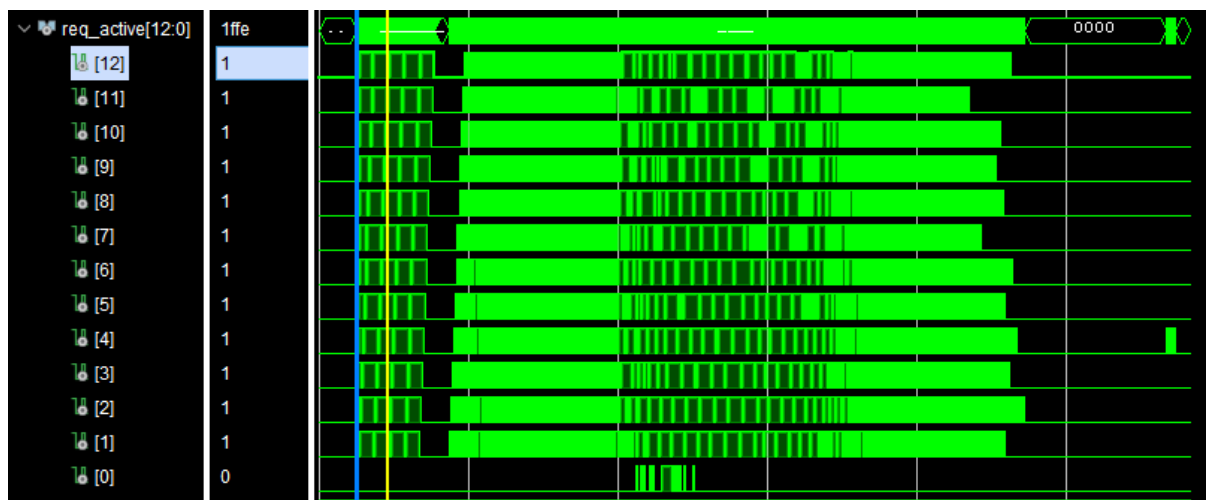


Figure 24: req\_active signal waveform

When configuring the router clock domain, note that it takes 8 clock cycles of the router clock for each data payload to “move” from requester to target FiFo. It takes 10 clock cycles (of SpaceWire Clock) for serialization of each data byte submitted to the CoDec Tx interface.

Therefore, in order to keep the SpaceWire link saturated, the router clock speed and data width must be configured so that the router fabric throughput is greater than the CoDec Tx throughput.

For example, with a data width of 4 and router clock speed of 200 MHz, the total data throughput would be 4 bytes every 40 ns. A SpaceWire port with 400MHz input clock would require 1byte every 25.0 ns. This configuration would satisfy the throughput as the average throughput of the router would be 1 byte every 10 ns.

However, if the router width was set to 1, then the throughput would be 1 byte every 40 ns. This would leave the Codec Tx-port to idle between byte transmissions. The router would bottleneck any transmissions across the network, which is undesirable.

As a rule of thumb, your router clock frequency should be configured as your SpaceWire port clock frequency divided by the fabric bus data width. Therefore, if operating a spacewire port at 400MHz and a data width of 2 is used, your router clock frequency should be 200MHz. Faster speeds can be used, but 200 MHz would guarantee the router fabric does not slow down any transmissions.

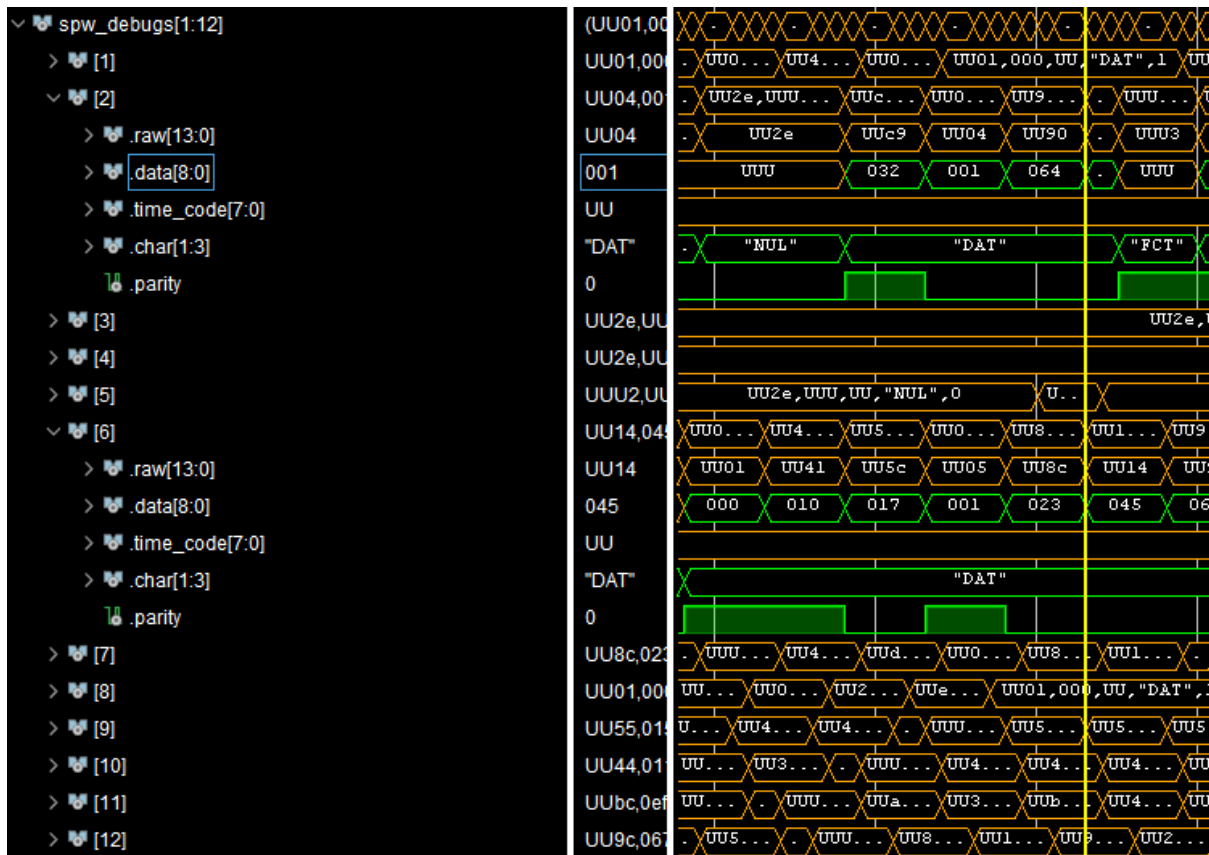


Figure 25: Simulation; SpW Debug Signals

The SpaceWire debug signals, *spw\_debugs* and *spw\_debugs\_in*, can be used to monitor the Tx and Rx SpaceWire links, respectively. These signals can be used to build verification procedures at the SpaceWire link-level for both behavioural and synthesis testbenches.

### 5.3 RMAP Simulation Procedures

In order to speed-up simulation using RMAP over SpaceWire a protected type has been created which facilitates the building of RMAP command headers.

The types *t\_rmap\_frame* and *t\_rmap\_frame\_array* is used to build RMAP commands for simulation. The latter type implements the *t\_rmap\_frame* as an array. It requires its own type as in VHDL 2008 protected types cannot be declared as arrays in signal/variable declaration. Ultimately both types have the same associated functions and procedures.

The *t\_rmap\_frame\_array* supports up to 32 RMAP frames.

The protected types are declared in *SpaceWire\_Sim\_lib.vhd* package file. See lines 822 and 1159 for the type-body declarations for the non-array and array versions, respectively.

The procedures *send\_rmap\_frame\_array* and *send\_rmap\_frame\_array\_raw* can be found on lines 2420 and 2468. These are used to send the RMAP frame data using the 4Links RMAP initiator and SpaceWire CoDec respectively. As implied in the name, these procedures use the *t\_rmap\_frame\_array* type as the data argument.

```

232 channel := i;
233 v_path_bytes(0) := c_num_ports - i;
234 rmap_frames.set_path_bytes(i, v_path_bytes);
235 rmap_frames.has_path_addr(i,true);
236 rmap_frames.set_logical_addr(i, 254);
237 rmap_frames.set_pid(i,1);
238 rmap_frames.set_instruction(i, "write", false, false, true, "00");
239 rmap_frames.set_key(i, 1);
240 rmap_frames.set_init_address(i, i);
241 rmap_frames.set_trans_id(i, 98+i);
242 rmap_frames.set_mem_address(i, 120+i);
243 rmap_frames.set_data_length(i, 16);
244 rmap_frames.set_data_bytes(i, c_data_test_pattern_1);
245 -- rmap_frames.set_header_crc(i);
246
247 wait until rst_in = '0';
248 wait for 25.4 us;
249
250 send_rmap_frame_raw_array(
251     channel,
252     codecs(i).Tx_data,
253     codecs(i).Tx_OR,
254     codecs(i).Tx_IR,
255     rmap_frames
256 );

```

Figure 26: rmap\_frame testbench Example

Figure 26 shows an example of the **t\_rmap\_frame\_array** command being used. First the RMAP command is built by calling functions and procedures within the protected type. The argument “i” is the RMAP command number in the array. A for-generate loop is used to create multiple concurrent processes where “i” is the loop integer.

Once the command has been built, it is sent using **send\_rmap\_frame\_raw\_array**. In this testbench, the stimulus IO is connected to directly to SpaceWire CoDecs. CRCs and EOP are automatically calculated and inserted where required when using these procedures.

## 6.0 Omitted Features (Initial Release)

At the current time there are several features which are omitted from the OpenSource router IP. However, these features can be added by modifying the IP source file as required. Future updates of the Router IP are subject to change but there is no guarantee that these features are added as part of the official release.

### 6.1 SpaceWire Interrupt Registers

Like Time-Code registers, the SpaceWire Router specification released by ECSS denotes that that a router “may” contain at least one Interrupt register. This interrupt capability is not present in the current release of the router IP. Modifications to the SpaceWire Open-Source CoDec would be required along with the top-level router architecture, in order to add this capability.

### 6.2 Port Statistics Registers

Some commercial router implementations contain port statistics registers for traffic logging and monitoring. The logic for these registers should be added as required. This can be achieved by utilizing/expanding the misc status registers and modifying the Port Rx and TX controller modules.

## 7.0 Arbitration Priority Schemes

This section covers the optional arbitration priority schemes in detail. The underlying RTL can be modified to implement custom arbitration priority schemes. The priority schemes discussed here are relevant to the priority schemes used for requester-target arbitration for the X-Bar fabric controller.

By default, the router IP implements a fair round-robin arbitration scheme for assigning outstanding requests to targets. No priority is used, and all ports have fair arbitration access.

Whilst this is theoretically good, it can lead to ports becoming “unlucky” causing large time delays before a request is serviced by the arbiter. Instead, a more desirable form of arbitration may be first-in-first-out (FiFo). FiFo arbitration prioritises requests which arrived first, for arbitration. This means that the possibility of a request being blocked by newer and/or faster requests is reduced. The downside here is that during address conflicts, some ports may now have to wait longer in order to be serviced, however, the mean time-to-service for all ports is lower and more consistent. This allows for the construction of more reliable spacewire networks.

### 7.1 FiFo Arbitration Methodology

The FiFo arbitration method is based on a priority queue. A list of requesters and targets are submitted to a list. The **write pointer** is incremented for each addition to the list. The **read pointer** acts as the arbitration pointer for the list. The 0<sup>th</sup> element in the list is the priority element, the oldest outstanding request in the arbitration fabric. Each list element is served by the arbitration logic. If the list element contains conflicting target assignments to the current active target assignments (or to the priority element), then the **read pointer** is incremented to the next list element. The **read pointer** will wrap-around from N to 0 if required, where N is the total number of ports on the router.

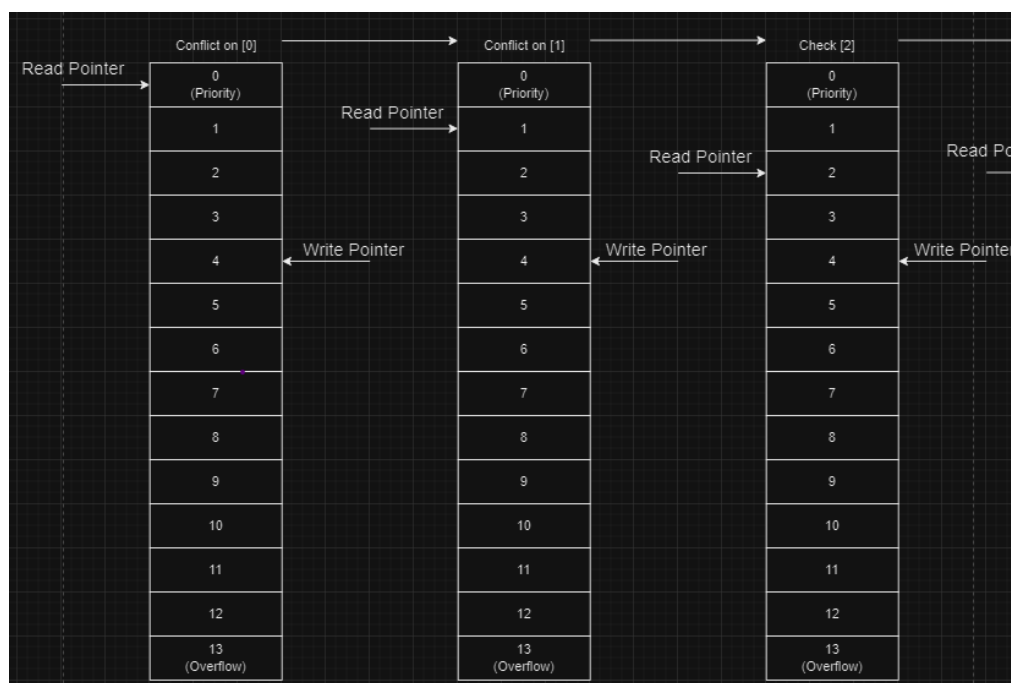


Figure 27:FiFo Arbitration: Checking Process

In the event a list element can be assigned with no conflicts, the targets for that element are assigned to the corresponding requester on the X-bar fabric. All elements below the current active



**read pointer** are shifted up one space. The current **write pointer** is also shifted up one space. The **read pointer** is then reset back to 0 and the process restarts.

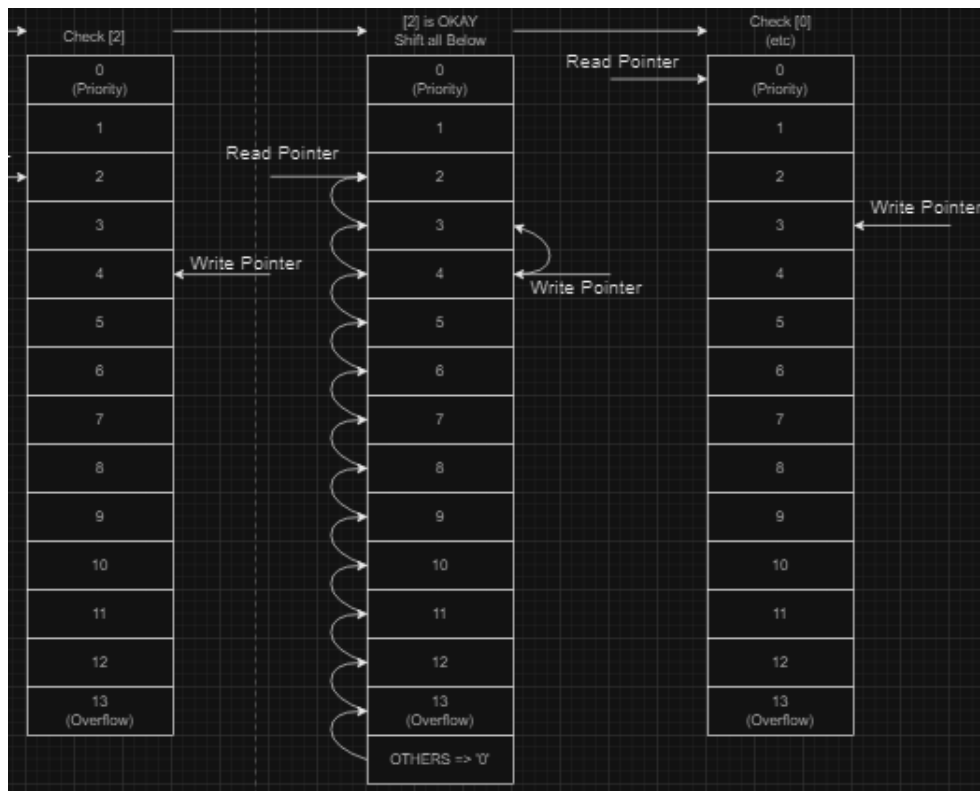


Figure 28: FiFo Arbitration: Shifting Process

This method of arbitration ensures that the oldest outstanding request (element 0) gets the most arbitration time. It also prevents newer requests from blocking the priority port by submitting conflicting target addresses. This feature is useful for when the priority request is a multi-cast target selection and lower-priority requests are single casts.

If the **read pointer** over-takes the **write pointer**, then the **read pointer** will wrap back to 0. All elements past the **write pointer** will contain all '0's. This is ensured by the design through the shifting-logic. Reading a list element past the **write pointer** will return all 0's and will cause no erroneous behaviour.

## 7.2 FiFo vs Round-Robin Architecture Comparison



Figure 29: FiFo based arbitration results

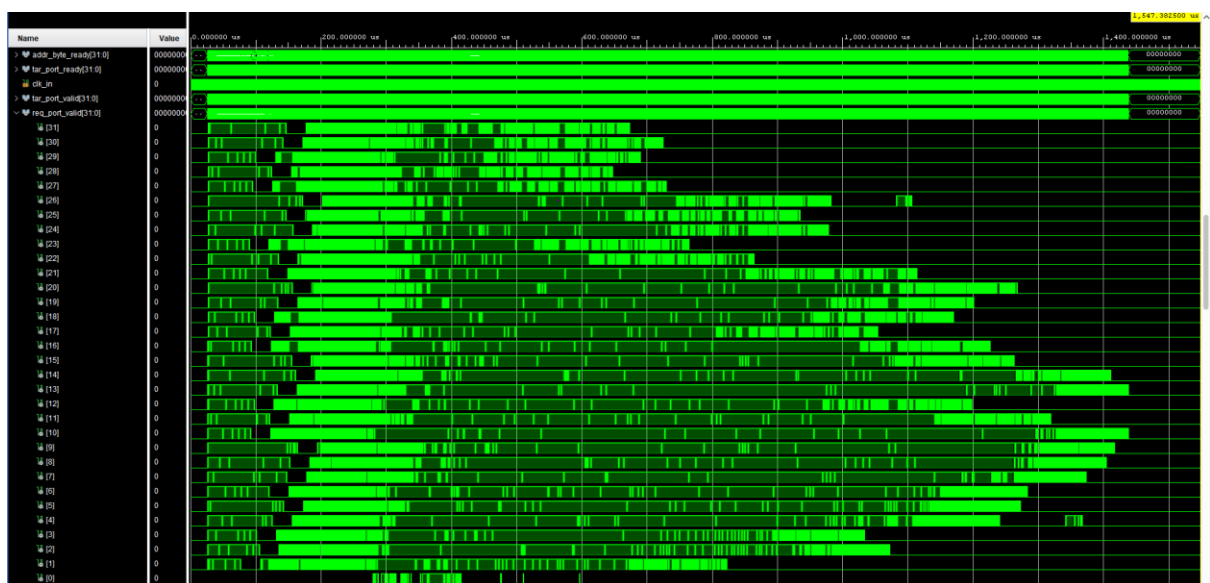


Figure 30: Fair round-robin arbitration results

The figures above show the same set of stimuli performed on a 32-port FIFO (Figure 29) and 32-Port round-robin based router (Figure 30). Note that for a FiFo based arbitration method, all physical ports complete their stimuli in around 1100 us.

However, when using Round-robin, there is a large variance in completion time. With the slowest port (13) completing at ~1440 us and the fastest port (28) at 650 us. This highlights the weakness of round-robin when using many physical ports.

For many applications, the FiFo based arbitration method provides a more desirable outcome, where latency across the router is much more predictable than round-robin. The differences between round-robin and FiFo based arbitration are exaggerated with high numbers of ports. For routers with few physical ports, round-robin based architecture may see no real latency penalty in comparison to the FiFo architecture.

## 8.0 Resource Utilization (Estimated)

### 8.1 12+1 ports (FiFo)

**Device:** XCKU040, -1 speed grade

SETTING	VALUE
SpW Port Frequency	400MHz
Router Frequency	200MHz
Fabric Bus Width	2x
Number of Ports	13 (12+1)
RAM style	"auto"

**Synthesis Attributes:** Vivado Synth Defaults

LUT	FF	BRAM
9245	10411	7.5

### 8.2 12+1 ports (Round Robin)

**Device:** XCKU040

SETTING	VALUE
SpW Port Frequency	400MHz
Router Frequency	200MHz
Fabric Bus Width	2x
Number of Ports	13 (12+1)
RAM style	"auto"

**Synthesis Attributes:** Vivado Synth Defaults

LUT	FF	BRAM
9389	11051	7.5

### 8.3 31+1 ports (FiFo)

**Device:** XCKU040

SETTING	VALUE
SpW Port Frequency	400MHz
Router Frequency	200MHz
Fabric Bus Width	2x
Number of Ports	32 (31+1)
RAM style	“auto”

**Synthesis Attributes:** Vivado Synth Defaults

LUT	FF	BRAM
27027	27499	18

### 8.4 31+1 ports (Round Robin)

**Device:** XCKU040

SETTING	VALUE
SpW Port Frequency	400MHz
Router Frequency	200MHz
Fabric Bus Width	2x
Number of Ports	32 (31+1)
RAM style	“auto”

**Synthesis Attributes:** Vivado Synth Defaults

LUT	FF	BRAM
27405	230881	18