SANDIA REPORT
SAND2008-5874
Unlimited Release
Printed September 2008

Hardware Demonstration of High-Speed Networks for Satellite Applications

Jonathon W. Donaldson and David S. Lee

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2008-5874
Unlimited Release
Printed September 2008

Hardware Demonstration of High-Speed Networks for Satellite Applications

Version 1.02
Last update: September 1, 2008

Jonathon W. Donaldson and David S. Lee
Wireless and Event Sensing Applications, 2664
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0986

ABSTRACT

This report documents the implementation results of a hardware demonstration utilizing the Serial RapidIO™ and SpaceWire protocols that was funded by Sandia National Laboratories' (SNL's) Laboratory Directed Research and Development (LDRD) office. This demonstration was one of the activities in the Modeling and Design of High-Speed Networks for Satellite Applications LDRD. This effort has demonstrated the transport of application layer packets across both RapidIO and SpaceWire networks to a common downlink destination using small topologies comprised of commercial-off-the-shelf and custom devices. The RapidFET and NEX-SRIO debug and verification tools were instrumental in the successful implementation of the RapidIO hardware demonstration. The SpaceWire hardware demonstration successfully demonstrated the transfer and routing of application data packets between multiple nodes and also was able reprogram remote nodes using configuration bitfiles transmitted over the network, a key feature proposed in node-based architectures (NBAs). Although a much larger network (at least 18 to 27 nodes) would be required to fully verify the design for use in a real-world application, this demonstration has shown that both RapidIO and SpaceWire are capable of routing application packets across a network to a common downlink node, illustrating their potential use in real-world NBAs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	Application Programming Interface
BRAM	Block RAM
BMP	bitmap
CAR	Capability Register
CCC	Command and Control
CCSDS	Consultative Committee for Space Data Systems
CF	CompactFlash
COE	coefficient
COTS	commercial off-the-shelf
CRC	cyclic redundancy check
CRF	Critical Request Flow
CSR	Command and Status Register
CTS	CCSDS to SRIO
CTSS	CCSDS to SRIO Solution
DAR	Device Access Routine
DCM	Digital Clock Manager
DLF	Downlink Framer
EDK	Embedded Development Kit
FF	Flip-Flop
FIFO	first in, first out
FPA	Focal Plane Array
FPGA	Field Programmable Gate Array
GSFC	Goddard Space Flight Center
GCLK	global clocking
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
IP	Intellectual Property
IRESP	Initiator Response
IREQ	Initiator Request
ISA	Instruction Set Architecture
ISF	internal switching fabric
ISO	International Standards Organization
LDRD	Laboratory Directed Research and Development
LFSR	linear feedback shift reporter
LUT	Lookup Table
LVDS	low-voltage differential signaling

MGT	multi-gigabit transceiver
NBA	Node-Based Architecture
OPB	On-chip Peripheral Bus
PCB	printed circuit board
PCI	peripheral component interconnect
PLB	Processor Local Bus
PPC	PowerPC
QoS	Quality of Service
RAM	random access memory
RCO	Receive Command Opcode
RCD	Receive Command Data
RCPM	Route Configuration and Port Monitoring
ROM	read-only memory
RREG	response register
SAR	Segmentation and Reassembly
SNL	Sandia National Laboratories
SRIO	Serial RapidIO
TID	Transaction ID
TREQ	Target Request
TUI	Textual User Interface
UART	Universal Asynchronous Receiver/Transmitter

EXECUTIVE SUMMARY

This report documents the implementation results of a hardware demonstration utilizing the Serial RapidIO™ and SpaceWire protocols that was funded by Sandia National Laboratories' (SNL's) Laboratory Directed Research and Development (LDRD) office. This demonstration was one of the activities in the Modeling and Design of High-Speed Networks for Satellite Applications LDRD [1].

The purpose of the research and development presented in this document was to demonstrate transport of application-layer packets across a network to a common downlink destination. In this demonstration the RapidIO and SpaceWire protocols were used as a conveyance for Consultative Committee for Space Data Systems (CCSDS) packets across a small network topology. The Serial RapidIO™ and SpaceWire protocols were chosen as possible candidates for network communications in node-based architectures (NBAs) for satellite systems in the Survey of Communication Protocols for Satellite Payloads [2]. The CCSDS protocol was chosen because SNL has a working history with the protocol, there was a previously written Hardware Description Language-based packet generator to leverage from, and it is a likely candidate for the application protocol used in future satellite architectures.

RapidIO is a commercial protocol that follows the standard Open Systems Interconnect networking model. The specification for RapidIO defines strict implementation directives for components spanning from the *physical* layer to the *transport* layer. A standard Application Programming Interface and function definitions are also provided for *application* layer designs interfacing to RapidIO hardware. RapidIO was chosen because its protocol specification allows for network scalability, guaranteed delivery, and ultra-high bandwidth. RapidIO is available in both serial and parallel physical layer implementations. The serial version of the physical RapidIO connectivity specification was chosen for this study because it requires the fewest connecting wires between nodes, consumes less power, allows for higher data rates, and results in less clock skew than its parallel counterpart.

SpaceWire is a bi-directional, full-duplex serial protocol developed primarily by the European Space Agency. SpaceWire is currently in use in a number of flight systems to provide a high-speed data infrastructure between sensors, processing elements, memory units, telemetry subsystems, and other space instruments [3]. As SpaceWire is already utilized in many space projects today, its feasibility for flight systems has already been established, making it a promising candidate for integration into networks for NBAs.

To adequately model real-world scenarios, the hardware demonstration assembled a representative model of real flight hardware using commercial-off-the-shelf development hardware. In typical flight systems, sensors would act as data generators and downlink modules as data sinks. The custom designs created for the hardware demonstration include a traffic generator node and a traffic sink node. The traffic generator nodes are responsible for encapsulating CCSDS packets in RapidIO frames or SpaceWire packets before transmitting them across the network. The traffic sink design consumes the RapidIO-encapsulated or SpaceWire CCSDS packets and reconstructs them for transfer to a CCSDS downlink framer. Integrating

these elements into the demonstration platform resulted in a successful demonstration of communication between nodes in a multi-node routed network.

One key desire in an NBA is the ability to dynamically reprogram endpoint logic in flight to provide different node functions within the network. This provides a number of advantages, including a failover capability to mitigate in-flight failures by reprogramming spare nodes to replace failed functionality. This node-based hardware demonstration was expanded to demonstrate the capability of reading a configuration bitfile from flash, transmit the bitfile over the network, and successfully reprogram remote nodes.

Although a much larger network (at least 18 to 27 nodes) would be required to fully verify the design for use in a real-world application, this demonstration has shown that both RapidIO and SpaceWire are capable of routing application packets across a network to a common downlink node, illustrating their potential use in real-world NBAs.

1. INTRODUCTION

The purpose of this study was to create a hardware design that demonstrated the ability to transfer application-layer packets from multiple source nodes across both RapidIO and SpaceWire network infrastructure to a common downlink destination node. The test network used in the demonstration was designed to use a small number of nodes based on commercial-off-the-shelf (COTS) development hardware as a proof-of-concept for a larger network topology.

From previous studies performed for this Laboratory Directed Research and Development (LDRD) [2], both Serial RapidIO™ and SpaceWire protocols emerged as potential candidates for use in NBAs. Since both protocols performed well in software simulation [4], the next step was to implement both Serial RapidIO and SpaceWire into hardware. To achieve this goal, a hardware demonstration built a representative model of the data flow that would be present in a real-world flight system. An additional goal of this demonstration was to characterize and validate the assessments performed in [2] of key protocol features. These observations and results are discussed throughout this document and cover features that include but are not limited to:

- Supported Bandwidths
- Overhead
- Latency
- Quality of Service (QoS)
- Fault Detection
- Reliability
- Error Correction
- Scalability

2. HARDWARE DEMONSTRATION OVERVIEW

Data flow in current architectures typically begins at sensor interface hardware. This hardware is responsible for collection of data from attached sensors that may generate large amounts of data. This data is shipped to data processor hardware, which may provide some level of in-flight data pre-processing and reduction. Finally, the processed data is sent to downlink channels for transmission to ground systems. Figure 1 portrays a high-level diagram of this concept.

To properly demonstrate the concept of a NBA, the components mentioned above were subdivided into distinct functions and separated into different node types. The approach to this demonstration was to provide enough hardware and development resources to demonstrate many functions of a real-world model.

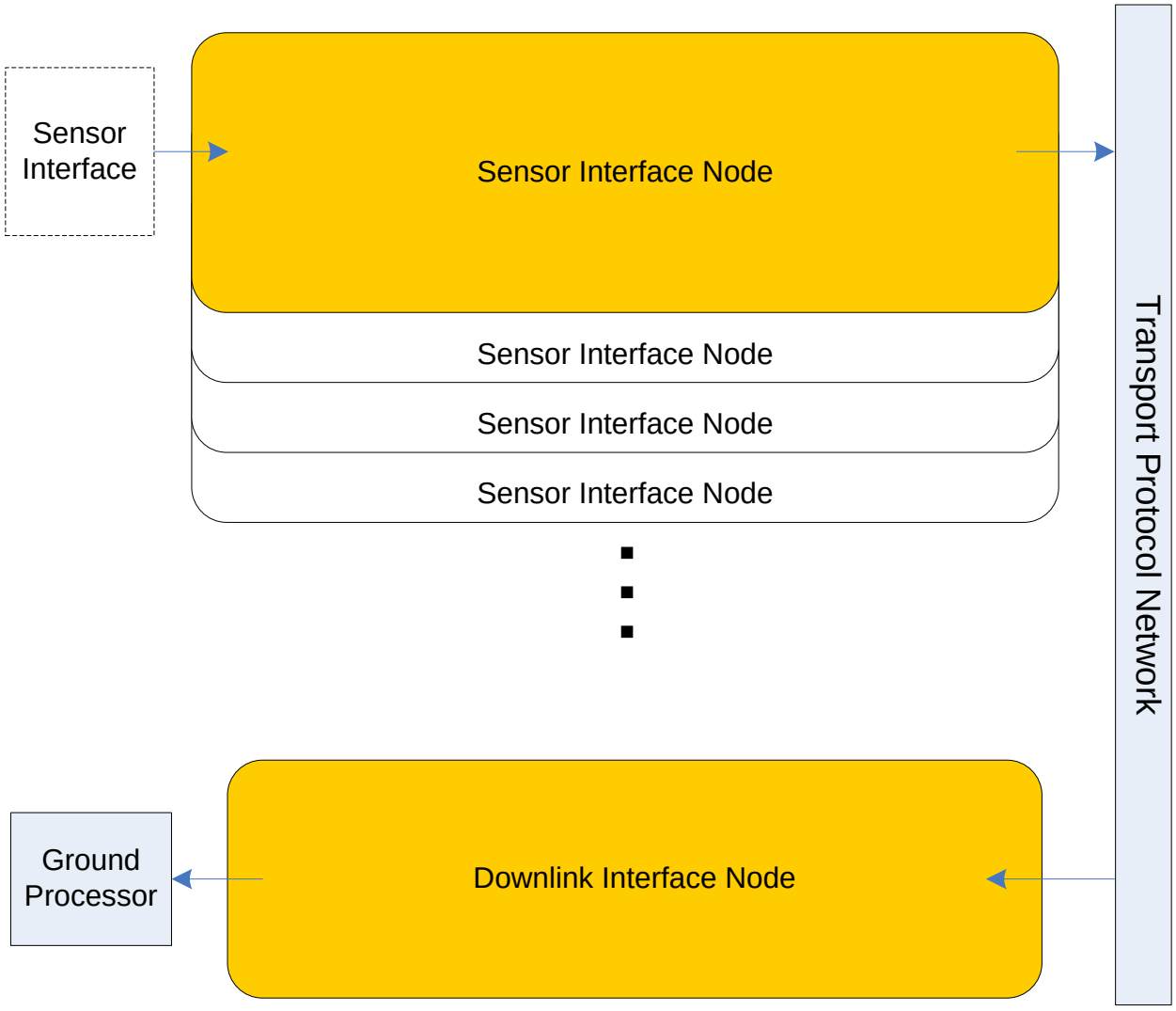

Figure 1. Basic block diagram of hardware demonstration.

2.1 Demonstration Components

The representative components in this hardware demonstration include a sensor interface node, a downlink or spacecraft communication node, a spacecraft interface, and a ground station. The sensor interface node is responsible for the generation of application data packets and transmits these data packets to the downlink node. The downlink node packages the data packets from all sensor node sources into frames and transmits these frames to the ground station, which can then analyze, display, or post-process the data.

Next, these representative flight component functions were mapped onto available COTS hardware. Where possible, COTS software or readily available Intellectual Property (IP) cores were used in place of any custom development efforts.

Development of sensor interface nodes includes three primary components. An application data component provides the source data to act as incoming sensor data. A data encapsulation component then packages the data into Consultative Committee for Space Data Systems (CCSDS) packets. The protocol interface component utilizes the appropriate network protocol (Serial RapidIO or SpaceWire) to transmit the information onto the network. The hardware utilized for these node types was typically a Xilinx ML325 prototype board [5].

The downlink node is also comprised of three components, also typically implemented on a Xilinx ML325 prototype board. The network protocol component receives network packets from the network. Data reconstruction strips off any network protocol specific information to isolate the source CCSDS packet. The downlink framer component takes these CCSDS packets and injects them into fixed-length CCSDS frames. These frames are then sent to the ground station. The CCSDS framer is a necessary component to communicate properly with the ground station, providing frame synchronization information and encapsulated packet information.

The final piece of the hardware demonstration is the ground station, which is provided by a PC equipped with a commercially available PCI-X CameraLink interface card. In order to get very high-speed data transfer (5.44 Gbps) of CCSDS frames into the PC, a custom interface board (hereafter referred to as the "FIB" test board) was leveraged from another SNL program. This FIB test board, developed by Ray Byrne and Joe Lyle at SNL, is able to receive four high-speed serial channels. The data received from these channels is bonded together to form one logical high-speed data channel. The FIB test board takes the logically bonded channel data and outputs this data via CameraLink. It should be noted that CameraLink requires a fixed-length data field and that can be accommodated by CCSDS frames fixed in size at 2044 bytes, whereas CCSDS packets in this demonstration were variable length. The entire architecture flow is shown in Figure 2.

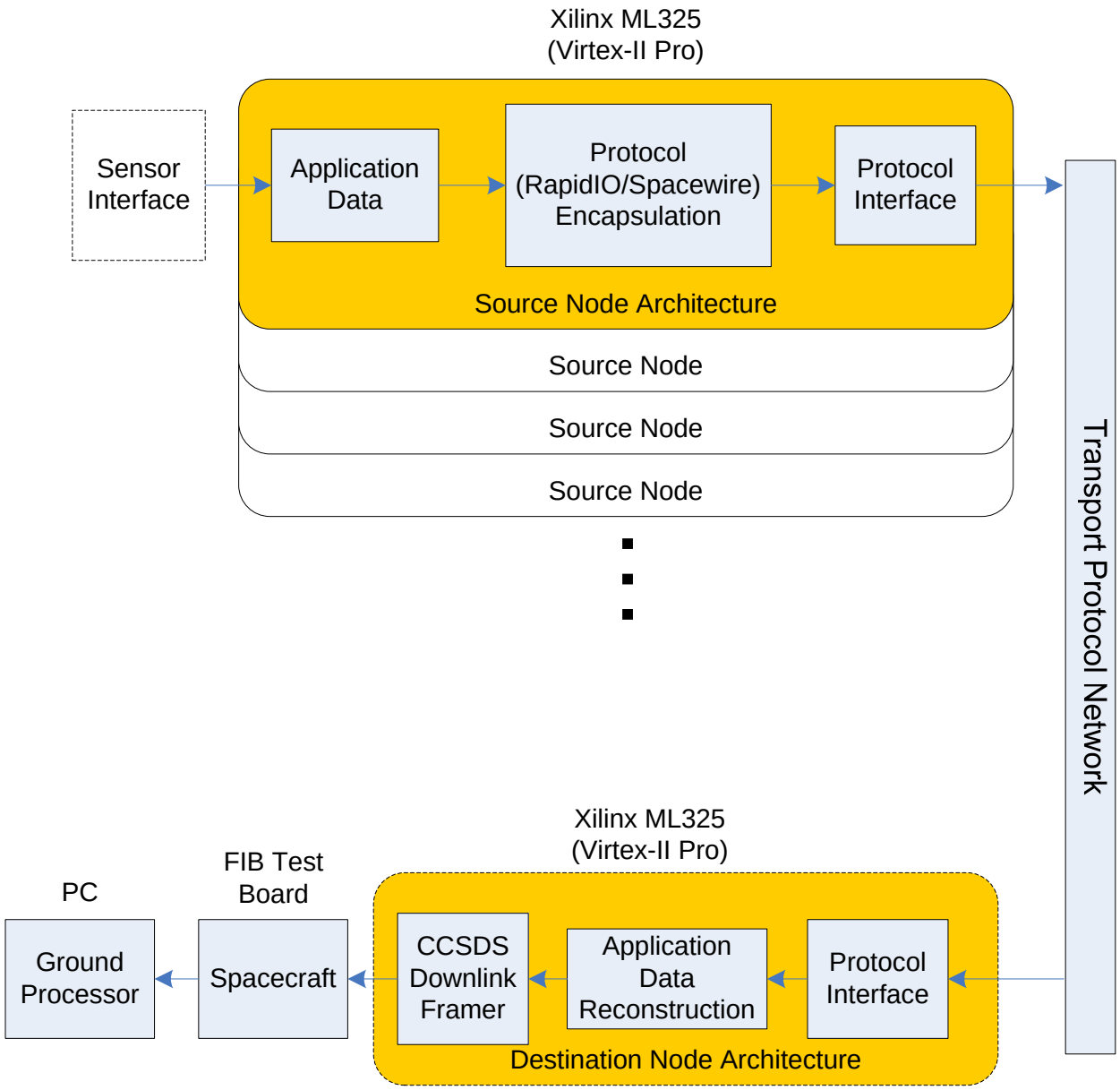

Figure 2. Detailed block diagram of hardware demonstration.

2.2 Hardware Implementation

The hardware demonstration incorporated a total of four nodes: three sensor interface nodes and one downlink node, which was connected to the ground station.

To introduce a visual component to the hardware implementation, a color image was taken and divided into its three constituent colors (red, green, and blue). Each sensor node was configured with one of the colors to serve as its sensor or "application data." The goal was to have each sensor node transmit its color component information through the downlink node to the ground station, where the image can be reassembled into a full-color image. If one of the nodes should

fail or if data transfer is interrupted for any reason, part or all of one color component will be missing and the image will be visually distorted. See Figure 3 for an example.

Figure 3. Progression of image data as each color source node is added to the network.

2.3 Additional Topics: Remote Reconfiguration

In addition to the demonstration of data exchange between nodes, another goal is to illustrate some of the other advanced features of NBAs. The ability to reprogram endpoint logic in flight to perform different node functions provides a number of advantages. Most notably, the ability to reprogram spare nodes with the functionality of failed nodes provides a failover mechanism that dramatically increases the reliability of the system. Thus, another objective of this hardware implementation is to demonstrate the ability to program nodes over the network.

3. RapidIO DEMONSTRATION

The discussion of the RapidIO demonstration begins with an overview of the test topology and is then broken down into two main sections: the source node design and the destination node design. The source nodes are responsible for generating the application layer packets while the destination node consumes them. The following sections provide a detailed overview of each component, both COTS and custom, which was utilized to implement the final solution. In addition, some useful design verification and debug tools are discussed that will aid in creating larger and more complex demonstrations in the future.

3.1 RapidIO Test Network Topology

The test network setup is shown in Figure 4. The system consists of four RapidIO endpoints connected via SMA cables to a centralized Tundra Tsi578 RapidIO switch. This topology was chosen not only because it simplifies the testing and debug process, but also because the flexibility of the RapidIO protocol allowed for verification of most traffic scenarios with a single switch. This flexibility is very advantageous because it allows for the development of a RapidIO system without requiring the development of switching IP that would be integrated into each individual node. In this case, the single switch was implemented using a COTS development board.

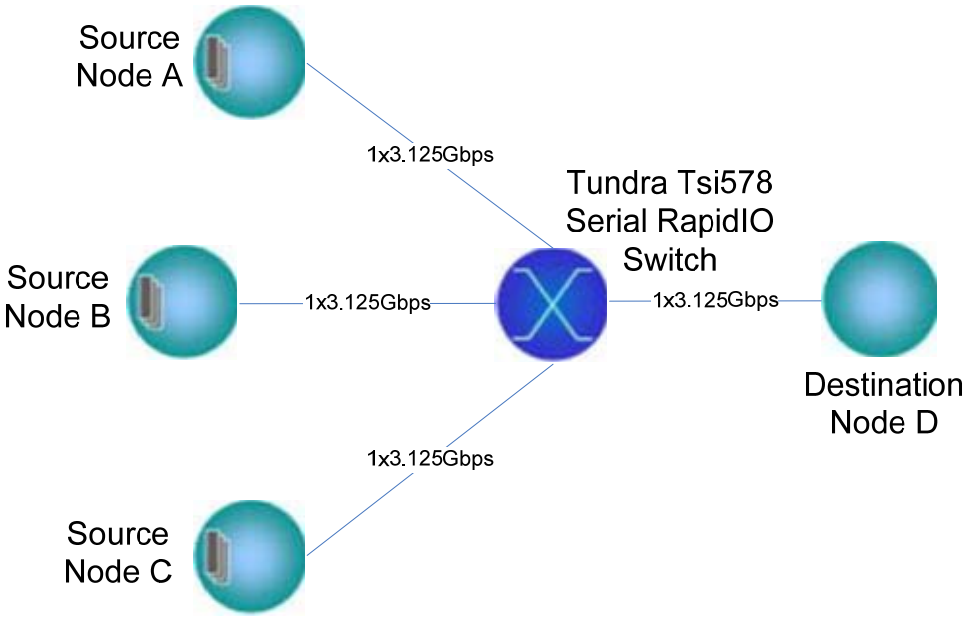

Figure 4. Test network topology.[1]

[1] The current revision of our design supports up to four source nodes; however, only three are shown here to simplify the diagram.

Unlike the SpaceWire demonstration (discussed in Section 4), there is presently no commercially available RapidIO switch IP that can be integrated into each source/destination node without developing a custom printed circuit board. Therefore, this demonstration was limited to the use of a fixed-Application-Specific Integrated Circuit switch development board that contained a Tsi578 Serial RapidIO switch from Tundra Semiconductor. This demonstrates a centralized switching topology rather than a distributed switching topology as discussed in Section 4. In order to scale a centralized switch topology, any Tsi578 switch can be directly attached to the port of another Tsi578 switch in order to increase the port count. The switches can be chained together indefinitely so long as the final endpoint node count does not exceed the RapidIO maximum network size set forth by the "Transport Type" field of each interacting node. The *transport type* field identifies whether a given endpoint supports either 8-bit or 16-bit device IDs.

The three nodes on the left are traffic sources and the node on the right is a traffic sink. Each node is attached to a different switch port using a 1x3.125 Gbps link. Factoring in the required 8B/10B data encoding used by RapidIO, this link speed allows for a theoretical maximum throughput of 2.5 Gbps. The RapidIO Version 1.3 physical layer specification allows links up to 4x3.125 Gbps for a total maximum throughput of 10 Gbps. It is also worth noting that the RapidIO switch will automatically detect the link rate (i.e., 1x/4x) of any node that is attached to it and synchronize the physical layers without any manual user intervention.

The Xilinx physical layer IP core does support the maximum allowable link rate; however, the purpose of this study was not to test the bandwidth capabilities of RapidIO. Therefore, a 1x link was chosen to simplify the design, reduce Field Programmable Gate Array (FPGA) resource usage, reduce implementation time, and reduce the number of physical cables needed in the system.

Though the RapidIO protocol does not specify a standard connection interface, the two cable types used in this system were CX4 (Infiniband) and coaxial with SMA connectors. Maximum cable lengths depend on the devices themselves and the data rate. Some devices have pre-emphasis on transmitters and equalization on receivers that extend their transmission lengths well beyond RapidIO specs of 22-inch backplanes. Moreover, the slower the data rate the longer the cable allowed. Certain vendors have demonstrated 4x 3.125 Gbps throughput across 10 meters of CX4 cable; however, typical applications implement CX4 lengths up to one meter and SMA coax lengths up to 28 inches.

3.1.1 Node Types

Each of the endpoints (A, B, C, D) utilize version 4.4 of the Xilinx Serial RapidIO Physical Layer Interface Core (part number: DO-DI-RIO-PHY) and the Xilinx RapidIO Logical and Transport Layer Interface core (part number: DO-DI-RIO-LOG)[2]. These two Xilinx cores are currently compliant with Version 1.3 of the official RapidIO specification.

The Xilinx RapidIO IP cores themselves are merely used as a conduit to interface and communicate with the physical Serial RapidIO network and by no means constitute the entire design on any of the endpoints, which would consist of application-specific activities. The

[2] At the time of this writing, neither of the Xilinx RapidIO IP cores is RIOLAB [6] certified.

remaining design will be discussed in later sections. The source and destination nodes were developed on a Xilinx ML325 development board fitted with a Virtex-II Pro XC2VP70.

At present, a single source endpoint (including the operating system and software interface) consumes approximately 15% of the V2Pro's internal logic resources, 20% of internal BlockRAM (BRAM),[3] and one of the two internal PowerPC 405 cores. The hard PowerPC core in the Xilinx was used in this design; however, a soft-core processor is another option. A processor is the preferred method of initializing and executing the mandatory RapidIO Application Programming Interface (API) functions discussed in Section 3.2.2 when the board is powered on. The processor is also used for the custom user interface discussed in Section 3.2.1.

The destination node consumes only 11% of the FPGA's logic resources and only 14% of the Block RAM space. The device utilization summaries from the Xilinx Place and Route tool are shown in Tables 1 and 2 for the source and destination nodes, respectively.

Table 1. Device Utilization Statistics for Serial RapidIO Source Node Design on Virtex-II Pro 70.[4]

Digital Clock Managers	2 out of 8	25%
Gigabit Transceivers	1 out of 20	5%
PPC405s	1 out of 2	50%
Block RAMs	67 out of 328	20%
Flip-Flops	9426 out of 66176	14%
4-input LUTs	11132 out of 66176	16%

Table 2. Device Utilization Statistics for Serial RapidIO Destination Node Design on Virtex-II Pro 70.[4]

Digital Clock Managers	1 out of 8	12%
Gigabit Transceivers	5 out of 20	25%
Block RAMs	48 out of 328	14%
Flip-Flops	7128 out of 66176	10%
4-input LUTs	8472 out of 66176	12%

[3] RAM – random access memory.

[4] These statistics reference the node implemented with the optional target scratch-pad memory (see Section 3.2.7).

Since the target device in the current processing architecture development is a Xilinx Virtex-5 FX130T, we can speculate as to the resource usage on a Virtex-5 FX130T for both the source and destination nodes from the Virtex-II Pro utilization reports. These estimations are shown in Tables 3 and 4, respectively. Furthermore, if the Xilinx physical layer Serial RapidIO (SRIO) core were re-generated to use 4x RapidIO links (instead of the present 1x configuration) it would consume approximately 4% more Lookup Tables (LUTs) and Flip-Flops (FFs) on either the V2Pro or the Virtex-5 architectures. A 4x configuration would also require an additional three sets of gigabit transceiver ports.

Table 3. Device Utilization Estimates for Serial RapidIO Source Node Design on Virtex-5 FX130T.[5]

DCMs	2 out of 12	16%
Gigabit Transceivers	1 out of 20	5%
PPC405s	1 out of 2	50%
Block RAMs	67 out of 596	11%
Flip-Flops	9426 out of 81920	12%
6-input LUTs	11132 out of 81920	13%

Table 4. Device Utilization Estimates for Serial RapidIO Destination Node Design on Virtex-5 FX130T.[5]

DCMs	1 out of 12	8%
Gigabit Transceivers	5 out of 20	25%
Block RAMs	48 out of 596	14%
Flip-Flops	7128 out of 81920	9%
6-input LUTs	8472 out of 81920	10%

The Tundra Tsi578 Serial RapidIO switch is built on to a development board from Silicon Turnkey Express [7]. The switch can be configured for a total of sixteen 1x link rate ports or eight 4x link rate ports. The board provides SMA, Infiniband, AMC, and various other High Speed Serial Interface (HSSI) connections for attaching to the Tundra switch ports. The switch development board requires a 20-pin ATX power supply; however, the board itself is not ATX form factor, which means that the board and power supply cannot be mounted inside a standard ATX chassis.

3.1.2 Traffic Flow Overview

Disregarding minor RapidIO handshaking frames, we can view nodes A, B, and C as the primary traffic sources and node D as the traffic sink. The three source nodes generate CCSDS packets

[5] These device utilizations are only estimates and are based solely upon the additional hardware resources available as per the Virtex-5 FX130T datasheet.

and RapidIO maintenance request/response frames when requested to do so by the user. While RapidIO maintenance transactions may occur between any two nodes, CCSDS packets are only sent to node D.

The RapidIO maintenance frames are needed to configure the endpoint and switch nodes subsequent to power on. Moreover, if any real-time changes need to be made to the switch's LUTs, or any interrupt/error status flags within any of the nodes need to be cleared, maintenance frames will be required. The power-on configuration steps include setting the endpoint/switch device IDs, setting the endpoint host lock ID, determining endpoint/switch state-of-health, configuring the switch LUTs, and initializing the switch's physical layer ports.

The CCSDS packets are first encapsulated into SRIO frames using the Message class (FType 11) before being dispatched from the source node. The Message class was chosen because it utilizes sequence ID numbers for each RapidIO frame sent. This allows RapidIO frames to be received in any order on the destination node while still allowing the full CCSDS packet to be properly reconstructed. This process is referred to as Segmentation and Reassembly (SAR).

The Message class also allows response frames to be sent back to the originating source node, thus creating an application-layer packet flow control mechanism. Aside from the built-in flow control of the RapidIO protocol, an additional CCSDS packet handshaking function was implemented that allows no more than one CCSDS packet to be in transit between all source/destination pairs at any time (see Section 3.3.1 for more information).

Each CCSDS packet is variable length with a maximum of 8188 bytes and constitutes one or more RapidIO frames. Each RapidIO frame has a maximum payload size of 256 bytes. Moreover, each RapidIO Message class frame requires an additional 8 bytes of protocol overhead. These 8 bytes consist of information specific to the physical, logical, and transport layers of the frame, which includes a 16-bit cyclic redundancy check (CRC) for error checking purposes (see Reference 8 for more information regarding these fields). This additional information results in a ~3% overhead for every CCSDS packet sent.[6]

In addition to overhead, packet transmission is also susceptible to the inherent latency within the switch. Tundra defines latency "as the time interval between the first bit of the Start-of-Packet arriving at the ingress of the Tsi578 and that same bit leaving the device" [9]. The cross-switch latency for each possible serial RapidIO link-rate is shown in Table 5.

[6] Note that this overhead calculation does not account for the necessary application-layer handshaking mechanism between the source/destination pair.

Table 5. Serial RapidIO 4x/1x Latency Numbers Under No Congestion (courtesy of [9]).

Reference Clock	Ingress and Egress Port Width	Ingress and Egress Baud Rate	Minimum Latency (ns)[a]
156.25MHz	4x mode	3.125	112
		2.5	128.8
		1.25	212.8
	1x mode	3.125	137.6
		2.5	160.8
		1.25	276.8

a. Due to the asynchronous ability of the clock frequencies within the device, the latency numbers can vary as much by two clock periods of 3.2ns.

Upon receipt of a CCSDS packet, the destination node will forward the packet on to the CCSDS Downlink Framer (DLF) module. The DLF encapsulates one or more CCSDS packets into a CCSDS Frame, which can store 2024 bytes worth of CCSDS packets. The final CCSDS frame data is then reformatted and sent off-chip to another board, which transfers the data over a CameraLink interface to a desktop computer. The frame data and statistics can then be viewed by the user via custom software.

The Tsi578's internal switching fabric (ISF) is non-blocking to all traffic provided the total ingress data flow to any single egress port does not exceed the egress port's outbound bandwidth. Clearly, in the topology shown in Figure 4, there is a bottleneck for traffic approaching the destination node D if any more than one of the source nodes is enabled and the sum of their traffic exceeds 2.5 Gbps. If this occurs, the switch will send "Packet Retry" control symbols back to the physical layer core of the source nodes. Any source node that receives one of these symbols will continue to retransmit the packet from its physical layer output packet buffer until the switch replies with a "Packet Accepted" control symbol.

Note that the Tsi578 also has full support for RapidIO priority-based quality of service (QoS) frame scheduling; however, all RapidIO frames sent by the source and destination nodes in the current version of this demonstration have their priority field set to zero. This was done only to simplify the design and debug process. Please see Section 3.9 for more information on this topic.

In part, this study was used to test RapidIO's "guaranteed delivery" mechanism to ensure that all packets sent across the network would reliably reach the intended destination. The only way to truly test this capability was to design a bottleneck into the network that would cause the switch to block packets. Not only does the bottleneck exercise the RapidIO protocol itself, but it also

aids in verifying proper switch and endpoint functionality when subjected to high levels of switch port congestion.

3.2 Source Node Design

The source node consists of both software and hardware components with the responsibility of encapsulating CCSDS packets in the RapidIO frame format for subsequent transmission across the switched RapidIO network. The following section discusses each component within the design, along with its specific purpose, in a top-down hierarchical fashion. A high-level block diagram of the source node design is shown in Figure 5. Also shown in this figure are the different clock domains that were required to execute this demonstration.

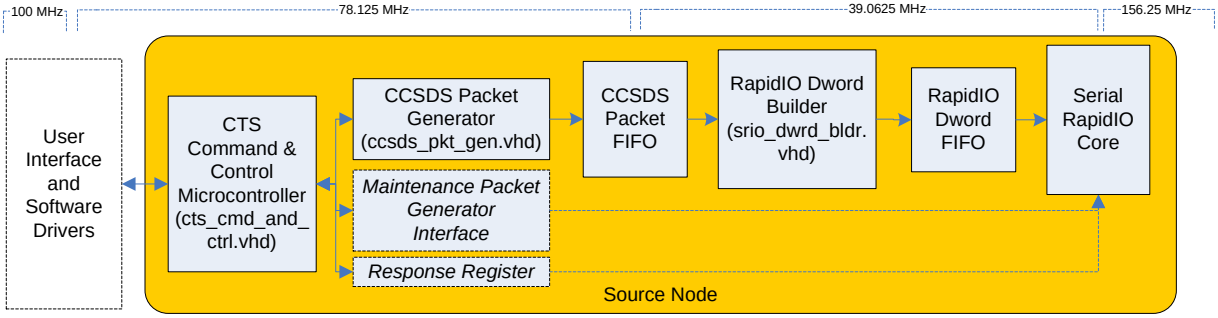

Figure 5. Block diagram of source node design.

3.2.1 User Interface

At the highest level of the design the user can interface to the Serial RapidIO hardware through a menu-driven textual user interface (TUI) over an RS-232 serial port. Presently, the menu options allow the user to send any number or size of CCSDS packets to the DLF node, read/write maintenance registers on any node, monitor switch state of health (e.g., congestion and interrupt registers), or populate the switch LUTs for the fixed topology described above. An example of the TUI is shown in Figure 6.

This small operating system is written in ANSI C and boots from internal Block RAM memory within the FPGA on power-up. The boot sequence is executed by the PowerPC, which initializes the hardware and creates a software-based instance of the device in main memory. The instance itself is a structure that contains information regarding the state of the hardware (e.g., base memory address, state of health, library initialization flags, etc.). After the boot sequence has completed, the RapidIO API library is then ready for input from the user.

Various debug options and output verbosity level parameters are also available. This debug information is very useful in diagnosing issues at the software/hardware layer interface. An example of this debug output is shown in Figure 7.

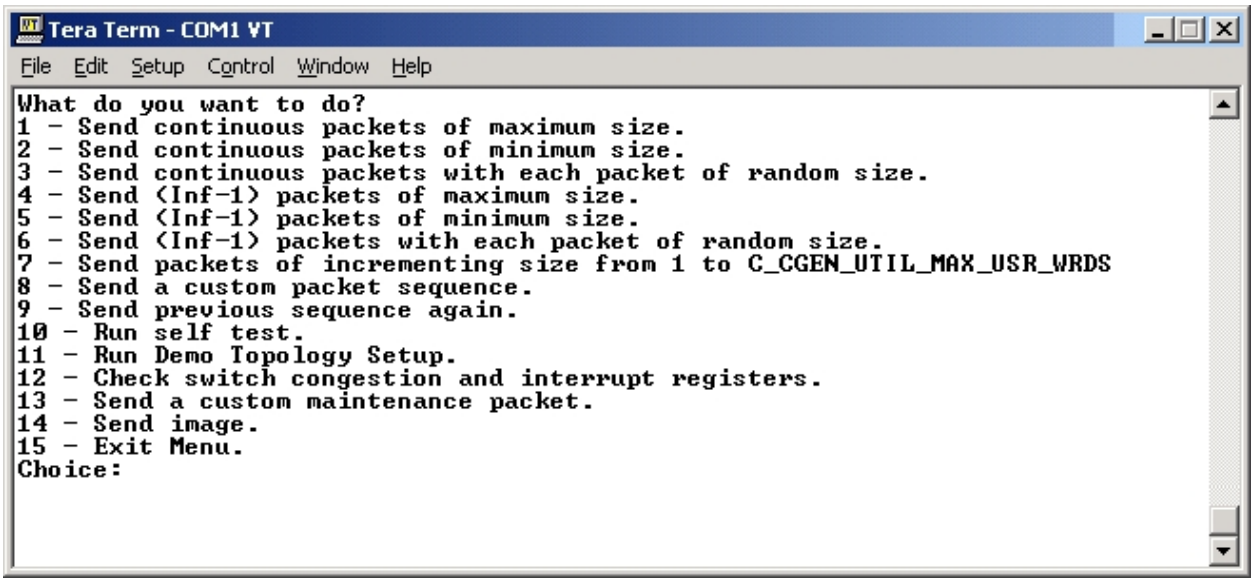

Figure 6. Textual user interface.

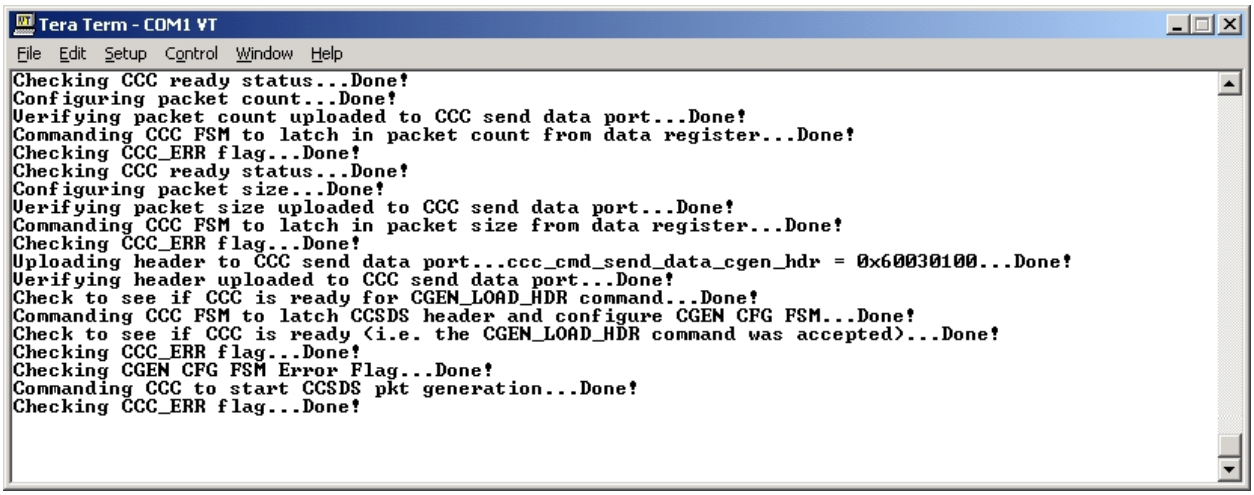

Figure 7. Example software-level debug output.

Each of the source nodes may contain up to two endpoints inside the single Virtex-II Pro FPGA on the ML325 development board. If two endpoints exist they are both physically and logically separate from one another – no "inter-node" communication is performed. The sole limitation when running two endpoints simultaneously on the ML325 board is that there is only a single Universal Asynchronous Receiver/Transmitter (UART) interface. To work around this problem, a UART multiplexer module was written that switches between the two endpoint STDIN/STDOUT interfaces depending on the state of an on-board Dual In-line Package switch.

An overview of the architecture used to interconnect the software and hardware components of a single endpoint is shown in Figure 8. All communications to and from the PowerPC are performed via the Processor Local Bus (PLB). The instruction and data memory are stored in the same Block RAM space with a single bus attachment because the PPC405 only has a five-stage pipeline and does not support out-of-order execution. The UART module is significantly slower than any other component in the system; therefore it is attached to the low-speed On-Chip Peripheral Bus (OPB) in order to simplify bus arbitration and reduce the number of wait states.

The CCSDS to SRIO Solution (CTSS) IP core that was designed for this study is directly attached to the high-speed PLB. This core, which includes all software and hardware components for the entire source node design, has been packaged into a complete Xilinx Embedded Development Kit (EDK) IP core that can be installed on any desktop PC and be viewed from the EDK IP Catalog library. The IP is packaged with Tcl [10] scripts that will automatically generate the required C-source code files that allow the core to be accessed in any custom design. This was done to make the core portable and easy to use by future designers. The interface to the core is discussed in detail in the following sections.

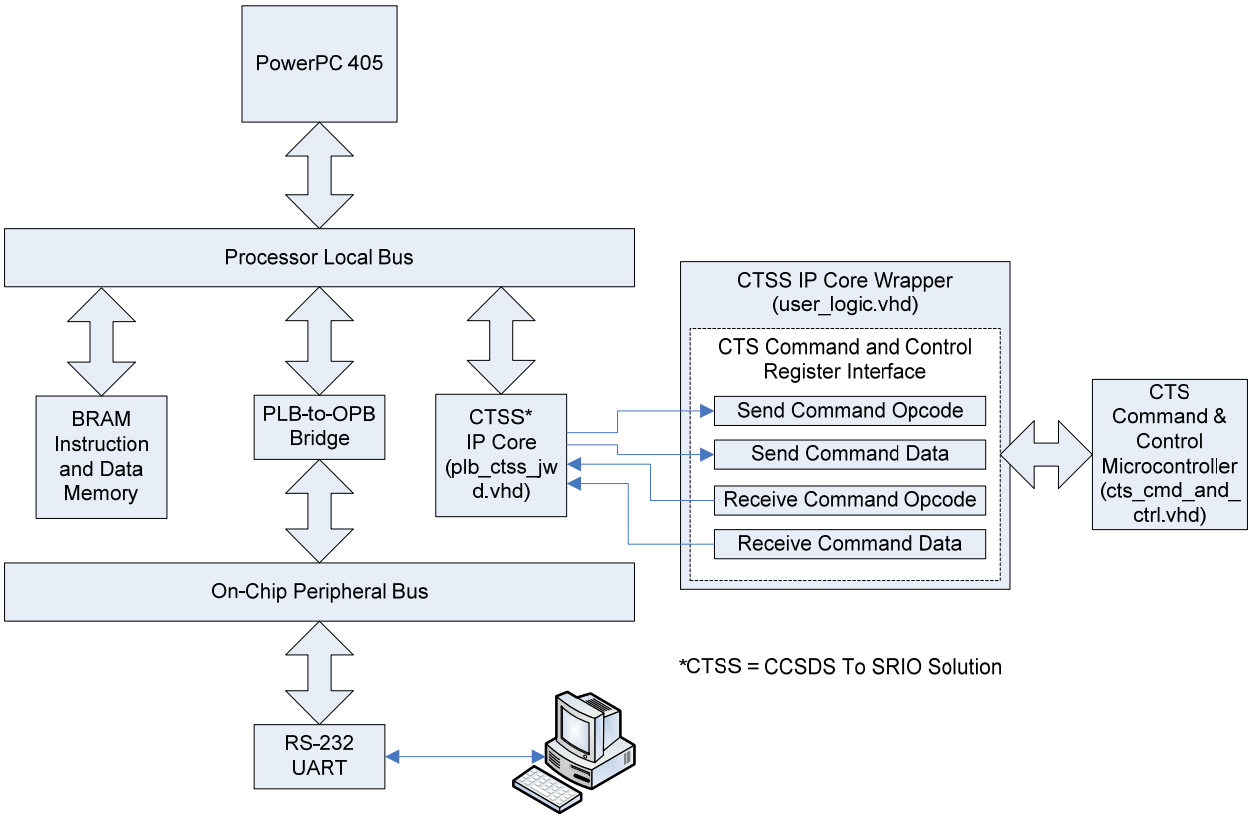

Figure 8. Software/hardware architecture.

3.2.2 Software API

When the user selects from one of the provided menu interface options, a series of API function calls are executed by the PowerPC in the background to generate the type of CCSDS packet or RapidIO Maintenance frame requested. The API used for the generation of RapidIO Maintenance frames is standard and is specified in "Annex 1: Software/System Bring Up Specification" of the official RapidIO specification. The API used for the generation of CCSDS packets is non-standard and was designed specifically for this study; however, it follows a similar function layout as the standard RapidIO API library structure.

In general, the API device drivers follow a layered architecture as shown in Figure 9. The "RTOS Adapter" functions are callable directly by the user whereas the "Device Driver" functions are meant to be accessed through the adapter functions only. The "Direct Hardware Interface" layer is accessed through the use of the IBM CoreConnect PLB and a Xilinx IP Interface module that allows for software accessible read/write registers within the FPGA's user-programmable logic space.

The RapidIO Maintenance frame generation API drivers consist of four distinct function sets: Hardware Abstraction Layer (HAL), Standard Bring Up, Routing-Table Manipulation, and Device Access Routine (DAR) Interface functions. The HAL functions can be considered Layer 1 functions while the other function sets are part of Layer 2.

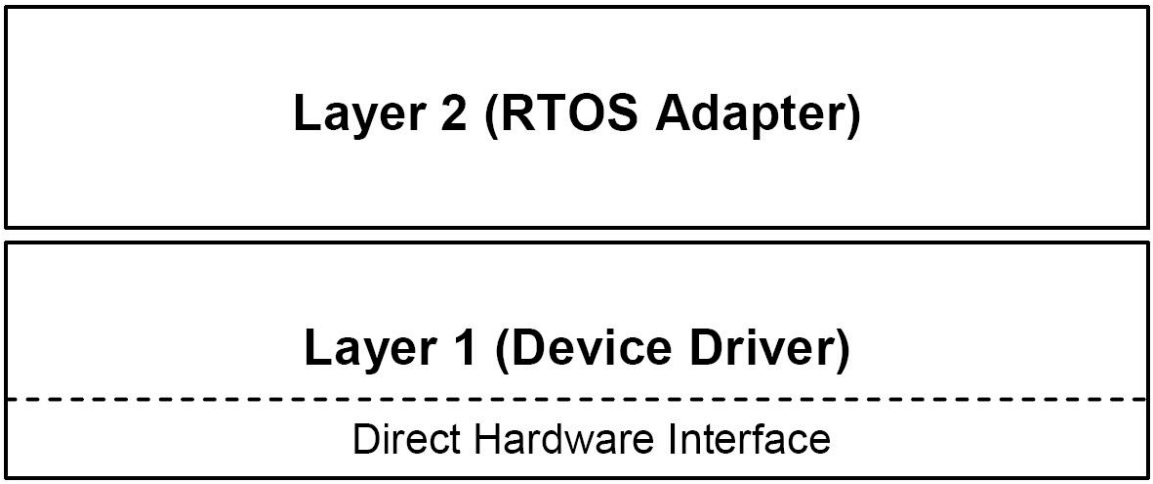

Figure 9. Software layers (courtesy Xilinx, Inc.).

There are only two substantial functions within the HAL set – one performs a maintenance read and the other performs a maintenance write. The other three function sets build on top of these two low-level driver functions and allow the user to configure endpoints/switches, monitor state of health, and configure the network topology by populating switch LUTs. At the time of this writing all function sets except for the DAR functions have been fully implemented.

The CCSDS Packet Generation API drivers control the hardware-based CCSDS packet generator module and allow the user to generate fixed or random size CCSDS packets, infinite/finite packet streams, and anything in between. The API functions include error checking, which will ensure that the user only generates CCSDS packets that are within the bounds of the official CCSDS specification.

In a real flight system, the CCSDS API functions would likely not be required, as a hardware interface to a sensor would be used to feed data directly into the CCSDS packet generator module. This would also eliminate the need for the UART and the TUI; however, the PowerPC (or equivalent soft-core processor) would still be required in order to run the aforementioned RapidIO maintenance configuration transactions (please see Section 3.2.5.2 for more information regarding processor requirements).

3.2.3 Hardware Interface

Immediately hanging off of the PLB are four software-accessible, 32-bit hardware registers. These registers allow the software to interface to a custom microcontroller that accepts opcodes and parameters. Opcodes sent from the software tell the microcontroller whether it should upload configuration data to the CCSDS packet generator module or the Maintenance frame generator module.

3.2.3.1 Send Command Opcode/Data Registers

Opcodes received from the microcontroller by the software inform the user if there is a RapidIO Maintenance response frame waiting to be read. The block diagram shown in Figure 8 refers to this module as the CCSDS to SRIO (CTS) Command and Control (CCC) microcontroller. The microcontroller's Instruction Set Architecture (ISA) presently consists of eight opcodes:

3.2.3.1.1 Opcodes for CCSDS Packet Generator

- **C_CGEN_LOAD_HDR** – This opcode is used to tell the microcontroller to latch the value in the "Send Command Data" register and set certain option flags and parameters within the CCSDS packet generator. These flags determine the packet generator's use of automatic coarse/fine time generation, the random/fixed packet size, the RapidIO frame priority to use, the RapidIO destination ID, and the RapidIO hop count to the specified destination. Note that the RapidIO parameters are stored in the CCSDS packet's Application Process Identifier field and used by another hardware module later in the packet pipeline.

- **C_CGEN_LOAD_PKTS** – This opcode is used to tell the microcontroller to latch the value in the "Send Command Data" register and upload it to the CCSDS packet generator's internal packet count register.

- **C_CGEN_LOAD_WRDS** – This opcode is used to tell the microcontroller to latch the value in the "Send Command Data" register and upload it to the CCSDS packet generator's internal packet size register.

- **C_CGEN_GO** – This opcode is used to tell the CCSDS packet generator to commence packet generation with the current input configuration. Note that the CCSDS packet generator module must have a valid configuration loaded at least once after power-up before any CCSDS packets can be generated.

3.2.3.1.2 Opcodes for RapidIO Maintenance Frame Generator

- **C_MGEN_LOAD_HDR** – This opcode is used to tell the microcontroller to latch the value in the "Send Command Data" register and set certain option flags and parameters within the Maintenance frame generator. These flags determine if the maintenance request is local/remote, the priority level, transaction type, destination ID, and hop count.

- **C_MGEN_LOAD_ADDR** – This opcode is used to tell the microcontroller to latch the value in the "Send Command Data" register and upload it to the Maintenance frame generator's internal offset register. The offset determines which RapidIO Capability Register (CAR) or Command and Status Register (CSR) will be read/written.

- **C_MGEN_LOAD_DATA** – This opcode is used to tell the microcontroller to latch the value in the "Send Command Data" register and upload it to the Maintenance frame generator's internal data register. This instruction needs to be executed only if the user is performing a maintenance write.

- **C_MGEN_GO** – This opcode is used to tell the maintenance frame generator to commence frame generation with the current input configuration. Note that the generator module must have a valid configuration loaded at least once after power-up before any frames can be generated.

3.2.3.2 Receive Command Opcode/Data Registers

When performing maintenance reads/writes the CCC module will also provide the maintenance response frame back to the software layer. Each time a maintenance request is generated by the user the software will poll the CCC module until the Receive Command Opcode (RCO) register contains the "Response Register Valid" instruction. If the response register (RREG) is valid, then the endpoint has received the corresponding maintenance response frame.

If the original request was a maintenance write, then the user need only read the "Receive Command Data" (RCD) register once to determine the status of the transaction (e.g., "Done" or "Error"). However, if the original request was a maintenance read, the user must first determine the status of the transaction and then read the RCD register a second time to fetch the maintenance response data.

In addition to informing the software layer of the receipt of a maintenance response frame, the RCO register also has instructions that inform the software of errors within various blocks along the packet/frame generation pipeline. The error opcodes are as follows:

- **C_CCC_ERR** – A recoverable error has occurred within the CCC module itself. This could be caused by an invalid instruction being sent to the CCC module by the user or due to an error in communication with the packet/frame generator modules.

- **C_CGEN_ERR_0** – An invalid packet sequence configuration has been uploaded to the CCSDS Packet Generator module. The user can recover from this error by uploading a valid configuration.

- **C_CGEN_ERR_1** – The CCSDS packet storage first in, first out (FIFO) has overflowed and data has been lost. This is an unrecoverable error for debugging purposes.[7]

- **C_MGEN_ERR** – An invalid frame configuration has been uploaded to the RapidIO Maintenance frame generation module. The user can recover from this error by uploading a valid configuration.

- **C_SRIO_BLDR_ERR_0** – The CCSDS packet storage FIFO has under-flowed and data has been lost. This is an unrecoverable error for debugging purposes.[7]

- **C_SRIO_BLDR_ERR_1** – The RapidIO Dword Builder module has received a CCSDS packet with an invalid format. This is an unrecoverable error for debugging purposes but will never be encountered by a standard user since the CCSDS packet generator module only allows the user to generate valid packets.

- **C_SRIO_BLDR_ERR_2** – The RapidIO Dword storage FIFO has overflowed and data has been lost. This is an unrecoverable error for debugging purposes.[8]

- **C_IGEN_ERR_0** – The RapidIO Dword storage FIFO has under-flowed and data has been lost. This is an unrecoverable error for debugging purposes.[9]

- **C_IGEN_ERR_1** – The Logical Layer RapidIO frame storage FIFO has overflowed and data has been lost. This is an unrecoverable error for debugging purposes.[10]

- **C_IGEN_MUX_ERR_0** – The maximum number of outstanding transactions (i.e., number of RapidIO response frames yet to be received) has been exceeded by the Initiator Request (IREQ) Multiplexer module. This is an unrecoverable error for debugging purposes and may be alleviated by increasing the maximum allowed number of outstanding transactions.[11]

---

[7] This error will never occur provided the source node IP created for this demonstration is used.
[8] Ibid.
[9] Ibid.
[10] Ibid.
[11] Ibid.

- **C_IGEN_MUX_ERR_1** – A sent RapidIO frame never received its required response. This is an unrecoverable error for debugging purposes.[12] The problem is likely with the original destination node of the original RapidIO request frame.

- **C_IHAND_ERR** – The Initiator Response (IRESP) Handler module has experienced an unrecoverable error. This error will occur if the IRESP Handler receives an unexpected response frame or an unexpected response frame format type. This error is only for debugging purposes.[13]

The RCO register also contains bits which can be used by the software to determine the empty/non-empty state of the TX and RX FIFOs in the packet/frame generation pipeline or the Xilinx core initialization status. The bit values are as follows:

- **C_RX_FIFO_STAT** – A value of 1 means the RapidIO Frame Receive FIFO is empty. A value of 0 means the RX FIFO is non-empty.

- **C_CGEN_FIFO_STAT** – Similar to C_RX_FIFO_STAT except for the CCSDS Packet Storage FIFO.

- **C_SRIO_FIFO_STAT** – Status of the RapidIO Dword storage FIFO.

- **C_LOGIO_FIFO_STAT** – Status of the RapidIO logical layer frame storage FIFO.

- **C_TX_FIFO_STAT** – Status of the transmit buffer FIFO, which sits between the Xilinx Logical and Physical Layer Serial RapidIO cores.

- **C_SRIO_CORE_STAT** – Status of the Xilinx Physical and Logical Layer cores (4 bits):

    - [3] – High if the physical layer has experienced no port errors. Low otherwise.
    - [2] – High if the physical layer has been properly initialized. Low otherwise.
    - [1] – High if the physical layer receive module is ready to accept data.
    - [0] – High if the physical layer transmit module is ready to accept data.

3.2.4 CCSDS Packet Encapsulation Pipeline

The CCSDS packet encapsulation pipeline consists of all VHSIC/Verilog Hardware Description Language (HDL) modules extending from the generation of the original CCSDS packet through to the emission of the actual SRIO frame from the Xilinx physical layer core.

[12] Ibid.
[13] Ibid.

3.2.4.1 CCSDS Packet Generator Module (ccsds_pkt_gen.vhd)

The CCSDS packet generator module can be configured by the user (through the CTS Command and Control module) to generate any number of CCSDS packets of any size. The configuration state machine within the module will also check to ensure that the user is only loading configurations that will produce valid CCSDS packet formats. If an invalid configuration is loaded, the module will assert an error back to the software layer to notify the user.

The module also has a built-in pseudo-random number generator that can be used to create packets of random size. The random numbers are created using a standard linear feedback shift register (LFSR). The user can choose whether or not to use random-sized packets by sending the C_CGEN_LOAD_HDR opcode to the CTS Command and Control module.

The coarse/fine time generation can also be adjusted to either insert coarse/fine time values from an external data port or to automatically generate a simple incrementing count which will reset at beginning of each new packet sequence. The automated time generation is very useful in debugging the packet transfer pipeline and for debugging received packets on the destination node.

Additionally, the user data within each CCSDS packet can be pulled in from an external data port (see Section 3.5 for an example) or can be automatically generated with a simple incrementing 16-bit count value. When using the automatic user data generation feature the count value will be reset for each new packet. The automated user data feature is very useful in debugging packet flows on both the source and destination nodes.

The CCSDS packet generator module also has a custom 32-bit RapidIO maintenance register (read-only) that can be accessed by the user through RapidIO maintenance read requests from any endpoint in the system. This register allows the user to ascertain the status of the packet generator module by providing information regarding the current state of all internal state machines, the number of user data words yet to be generated for the current CCSDS packet, and the number of packets yet to be sent for the current CCSDS packet sequence. This register is very useful in debugging the packet generator module.

3.2.4.2 RapidIO Dword Builder Module (srio_dwrd_bldr.vhd)

The RapidIO Dword Builder module converts 16-bit CCSDS packet words into 64-bit RapidIO dwords. The module pulls 16-bit CCSDS words from the CCSDS packet storage FIFO, packs them into 64-bit RapidIO dwords, and finally writes the dwords to the RapidIO dword storage FIFO. Any CCSDS packet that is not a multiple of 64-bits will have the unused 16-bit chunks within each dword “zeroed out” before being written to the dword storage FIFO.

The RapidIO Dword Builder module also has a custom 32-bit RapidIO maintenance register (read-only) that can be accessed by the user through RapidIO maintenance read requests from any endpoint in the system. The register allows the user to ascertain the status of the Dword Builder module by providing information regarding the current state of all internal state machines and the number of user data words yet to be converted for the CCSDS packet currently being processed.

The frame data from the Dword Builder module is stored in a FIFO, where it is eventually read out by lower level blocks in the Serial RapidIO Core design. A high-level block diagram of this core is shown in Figure 10. The components in this core are discussed in the following sections.

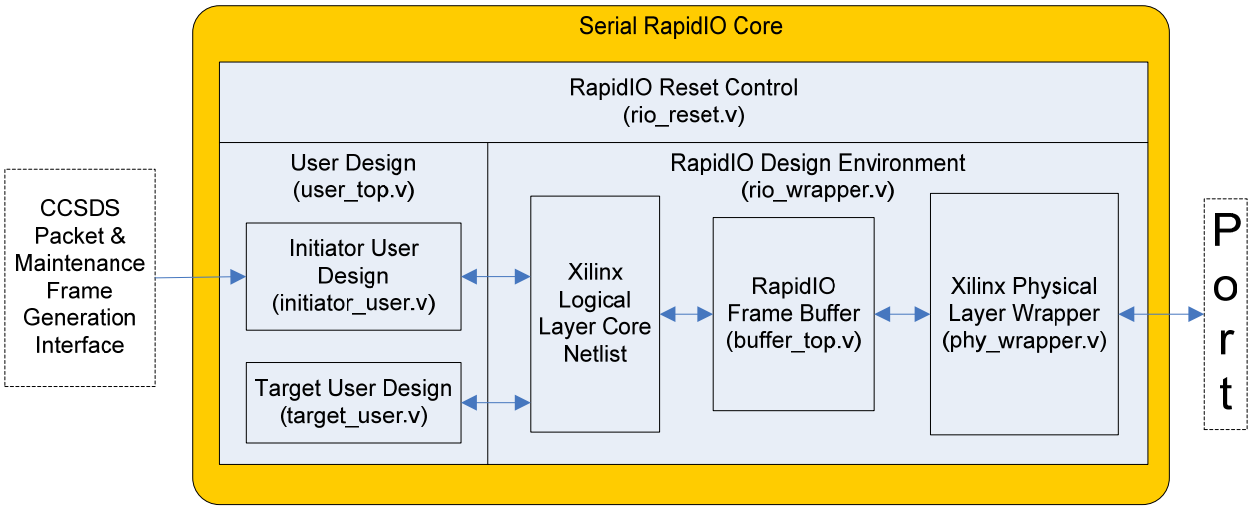

Figure 10. Block diagram of Serial RapidIO Core design.

3.2.4.3 CCSDS Initiator Request Generator Module (ccsds_ireq_gen.vhd)

The CCSDS Initiator Request (IREQ) generator is embedded inside the Initiator User Design block shown in Figure 10. A lower-level block diagram of the components within this module and the interconnections to the Xilinx Logical Layer core are shown in Figure 11. The CCSDS IREQ generator consumes the RapidIO dwords stored in the RapidIO Dword FIFO, encapsulates them into RapidIO Message class frames (i.e., FType 11), and finally forwards each frame onto the logical layer interface frame storage FIFO. Each RapidIO frame, except for (possibly) the very last frame, is the maximum allowed frame size of 256 bytes or 32 64-bit dwords.

RapidIO frame parameters such as the Critical Request Flow (CRF) bit, the frame priority, the destination ID, and hopcount are pulled from the CCSDS packet's API field during frame generation. Additionally, the RapidIO message length (msg_len) and message segment (msg_seg) identifier fields are automatically generated by an internal state machine.

It is important to note that the RapidIO Message class has a flow/sequence size limit of 4096 bytes due to the 4-bit msg_seg/msg_len fields. SNL's implemented CCSDS specification allows for packets up to 8188 bytes in size (including PHDR, SHDR, and CRC). This size difference poses a problem since it is not possible to encapsulate any CCSDS packet over 4096 bytes in size within a single RapidIO Message flow. In order to resolve this issue, an optional 2-bit field, referred to as the Mailbox bits, within the RapidIO Message frame type were used to add two more bits of resolution to each Message flow, thus allowing a maximum CCSDS packet size of 16384 bytes.

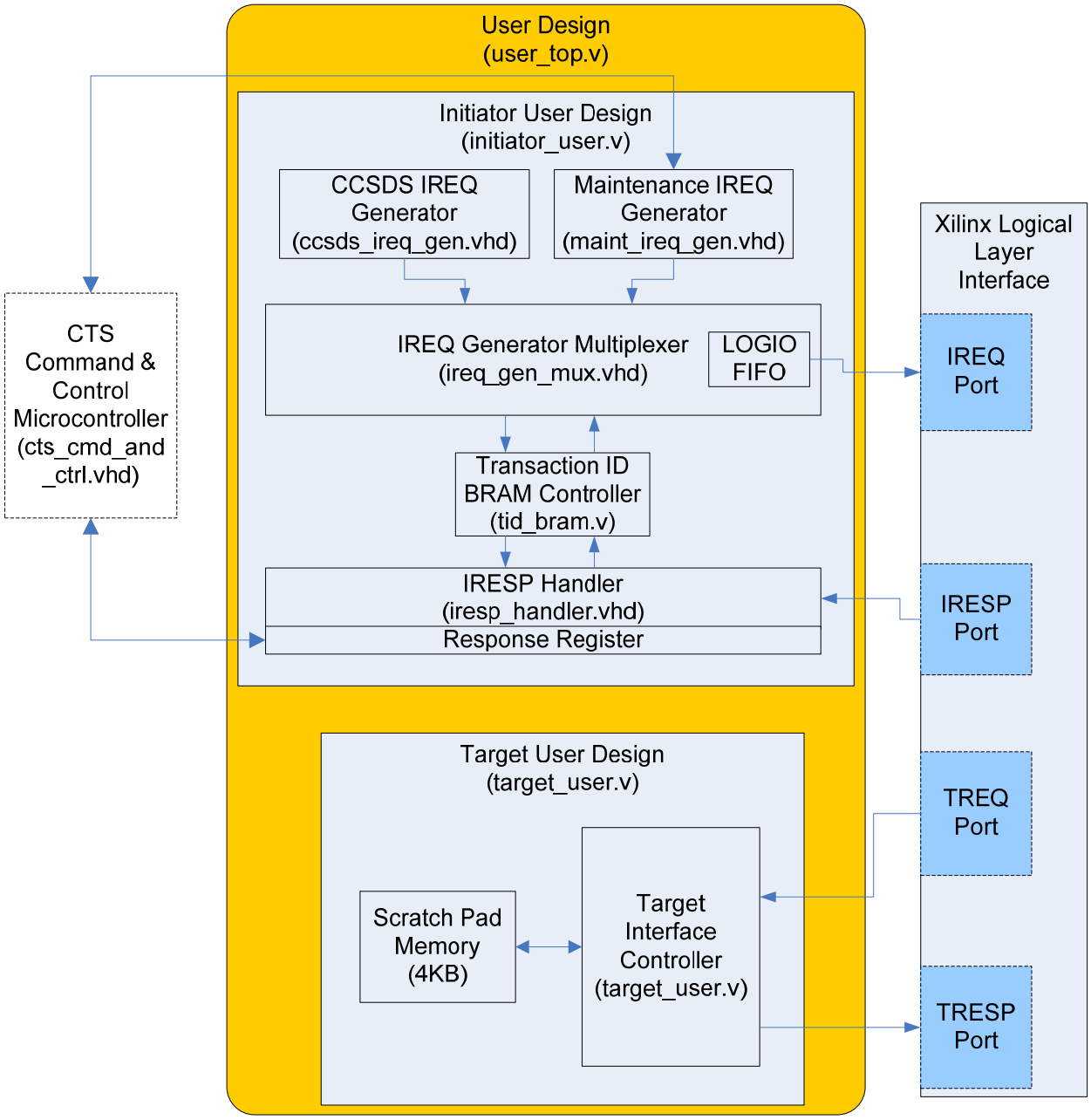

Figure 11. User design internal components.

Similar to the previous modules, the CCSDS IREQ generator module contains a custom 32-bit RapidIO maintenance register (read-only) that can be accessed by the user through RapidIO maintenance read requests from any endpoint in the system. The register allows the user to ascertain the status of the IREQ generator by providing information regarding the current state of all internal state machines, the number of RapidIO dwords yet to be sent to the LOGIO FIFO for the current CCSDS packet being processed, and the number of dwords yet to be sent to the LOGIO FIFO for the current RapidIO frame being processed.

3.2.4.4 Initiator Request Multiplexer Module (ireq_gen_mux.vhd)

The IREQ Multiplexer component (also shown in Figure 11) accepts requests from the CCSDS IREQ Generator module and the Maintenance Frame Generator for access to the LOGIO FIFO. Access to the LOGIO FIFO is based on a multi-channel "Request"/"Grant" handshaking mechanism. This module is necessary because there is only one IREQ port on the Xilinx Logical Layer core but two modules required access to the logical layer in order to send frames over the network. Therefore, it was necessary to design a module that would arbitrate access between the two transmitters.

In addition to controlling IREQ port access, the IREQ Multiplexer module also updates the transaction ID field for each RapidIO maintenance frame that is sent. Note that the Transaction ID (TID) is not incremented when sending RapidIO frames for CCSDS packets since the RapidIO Message class uses the "msg_seg" field for identification purposes.

For any sent RapidIO frames requiring a response, this module is also responsible for validating those frames inside the TID Block RAM. This module will also flag an error if the TID Block RAM module exceeds the maximum allowed number of outstanding frames. Additionally, this module ensures that all required responses are received for any frames sent. If either of these errors occurs they are reported back to the software layer.

3.2.5 Maintenance Frame Generation Pipeline

The RapidIO maintenance frame generator pipeline consists of all VHSIC/Verilog HDL modules extending from the generation of the original RapidIO maintenance frame through to the emission of the actual SRIO frame from the Xilinx physical layer core. A block diagram of the components in the pipeline and its interface to the CTS Command & Control Microcontroller is shown in Figure 11.

3.2.5.1 Maintenance Initiator Request Generator (maint_ireq_gen.vhd)

The Maintenance IREQ Generator component shown in Figure 11 generates RapidIO maintenance class (i.e., FType 8) read/write requests to offsets specified by the user through software layer functions. This module's configuration interface is almost identical to that of the CCSDS IREQ Generator module. The user may read/write any 32-bit maintenance register on any node attached to the network. Double-word (i.e., 64-bit) transactions or transactions less than 32-bit are not supported.

Since it is impossible for RapidIO to send anything less than 64-bit data chunks within a single frame, the IREQ Generator Multiplexer module will store the original offset address used for the maintenance request inside the TID Block RAM. This offset can then later be used when a corresponding read response is received to determine which half of the double-word quantity was requested by the user.

The Xilinx RapidIO cores support 34-bit addressing; however, the current version of the maintenance frame generator only supports 32-bit addresses. The upper two bits represent the

Extended Address Most Significant Bits field, which is noted in the official RapidIO specification.

3.2.5.2 Maintenance Request Generation Without Processor

A processor is the preferred method of generating maintenance transactions; however, if no processor option is available it would be possible to generate a finite set of maintenance commands to various nodes upon startup using an HDL-only implementation. This could be accomplished by storing a pre-defined list of configuration opcodes inside a read-only memory (ROM) that would be read by the CTS microcontroller (see Section 3.2.3) upon boot and sent to the maintenance frame generator module.

An implementation such as this works fairly well for static systems where the topology is fixed and known before power-on; however, it is very limited and can become extremely complex when considering real-time generation of dynamically configured maintenance frames. For example, RapidIO endpoints and switch cores contain numerous status registers, some of which are interrupt/error flags. Some of these flags must be cleared (using maintenance transactions) whenever they are set for continued proper operation of the node.[14]

3.2.6 Transaction ID Block RAM (tid_bram.v)

The TID Block RAM module shown in Figure 11 stores a valid/invalid history of all RapidIO frames sent from the source node that are expected to receive a corresponding RapidIO response frame. The only two frame types the source node sends that require a response are Message class frames (for CCSDS packets) and Maintenance class frames.

The Block RAM is logically separated into two distinct but equal memory spaces. One half of the memory is used for Message class frames and the other half is used for Maintenance frames. This is necessary since the Message and Maintenance classes use different fields for frame identifiers (i.e., the msg_seg field or the TID field).

On power-up this module will also invalidate (i.e., clear) all TID Block RAM locations before allowing any RapidIO frames to be sent. This prevents possible corruption of the TID memory space.

3.2.7 Scratch-Pad Memory Module (target_user.v)

The source node can be synthesized to either ignore or accept Target Requests (TREQs) from another device on the network. If the source is implemented to accept TREQ frames then frame types 2 (NREAD), 5 (NWRITE), or 6 (SWRITE) may be used to target the source node to test memory reads/writes across the RapidIO network. This module was provided by Xilinx in the reference design included with the core when it was purchased. The scratch-pad memory area is 4 Kbytes in size and is stored in the FPGA's Block RAM.

[14] An HDL-only implementation for maintenance transactions was not considered for this demonstration and is beyond the scope of this document.

It is important to note that if the source node is implemented to ignore TREQs then it will appear as if it is unresponsive, since no RapidIO response frames will be sent. The location and interface to the Target User Design module is shown in Figure 11.

3.2.8 Initiator Response Handler (iresp_handler.vhd)

The IRESP handler module accepts response frames to previously sent message or maintenance class request frames and invalidates the entry for that frame within the transaction ID Block RAM module. Please reference Figure 11 for a block diagram of this module, which includes its interconnections to the Xilinx core and its interface back to the software layer.

When all message class frames for a CCSDS packet have been received, the IRESP handler will report the event to the CCSDS IREQ Generator core, which will allow it to transmit another CCSDS packet.

For maintenance response frames the IRESP handler will extract the appropriate information from the frame and place it in the Response Register so that it may be read by the software layer through the CTS Command & Control module.

3.2.9 RapidIO Design Environment (rio_wrapper.v)

The RapidIO Design Environment shown in Figure 12 is a Verilog wrapper originally provided by Xilinx when the core is purchased. The wrapper encapsulates both the LOG and PHY layers and also incorporates a store and forward frame buffer design. Xilinx does not require the use of its frame buffer in order to properly operate the Xilinx cores; however, if a custom frame buffer is used it must be of the "store-and-forward" type, as the Xilinx Logical Layer core does not support source side stalls from the PHY.

All of the components within the RapidIO Design Environment are included with the physical and logical layers cores when they are purchased from Xilinx. A few minor modifications to the frame buffer reference design and patch for the physical layer netlist (both available from Xilinx, Inc. [11]) are required for proper operation of the Xilinx RapidIO endpoint in a switch environment. If these patches are not applied to the design the frame transmission pipeline to the PHY layer will "freeze" if too many "Packet Retry" symbols are received from the switch. SNL was instrumental in finding and researching this design flaw that allowed Xilinx to create the final patch. The physical layer patch is now incorporated in the Xilinx cores beginning with Version 4.4.

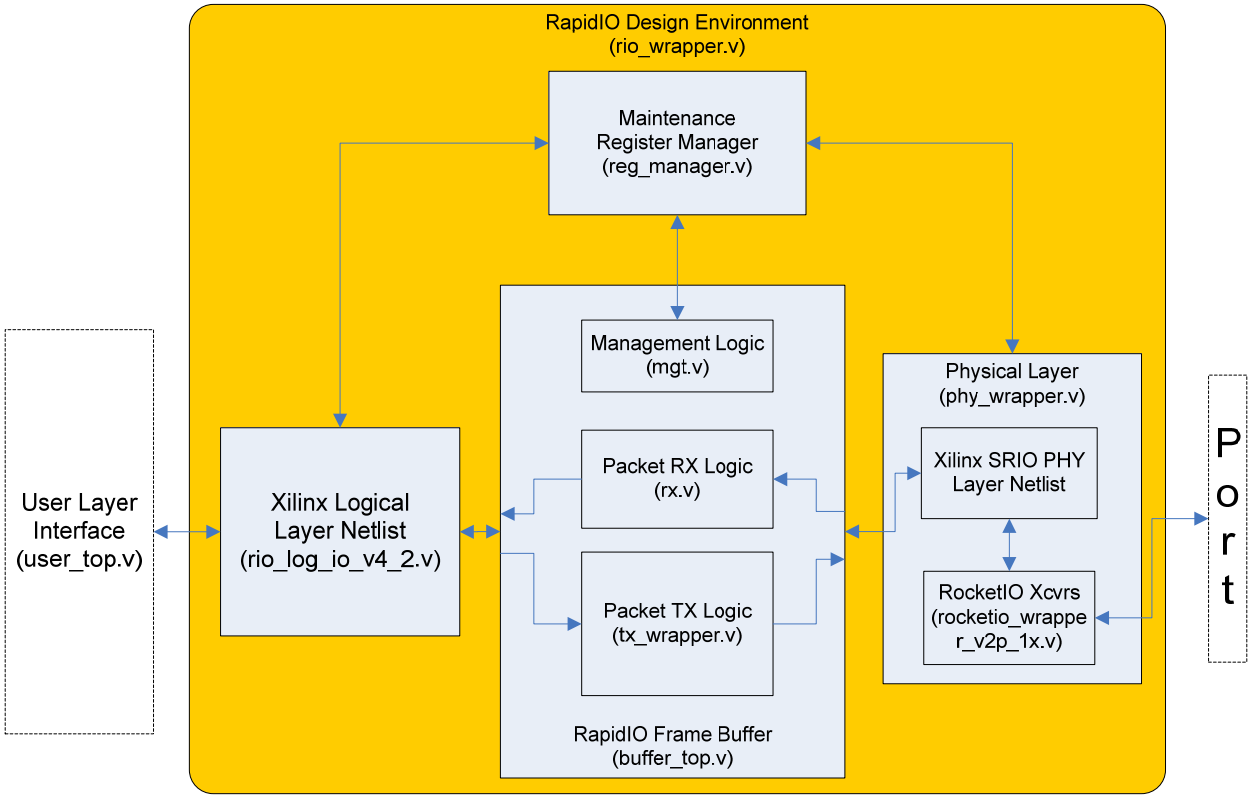

Figure 12. RapidIO Design Environment.

3.3 Destination Node Design

Unlike the source node design, the destination node has no software component or external user interface. All CCSDS packet and SRIO frame processing is performed strictly in hardware. The destination node consumes CCSDS packets and RapidIO Maintenance frames and generates the appropriate response frames. A high-level block diagram of the destination node architecture is shown in Figure 13. Also shown in this figure are the different clock domains that were required to execute this demonstration.

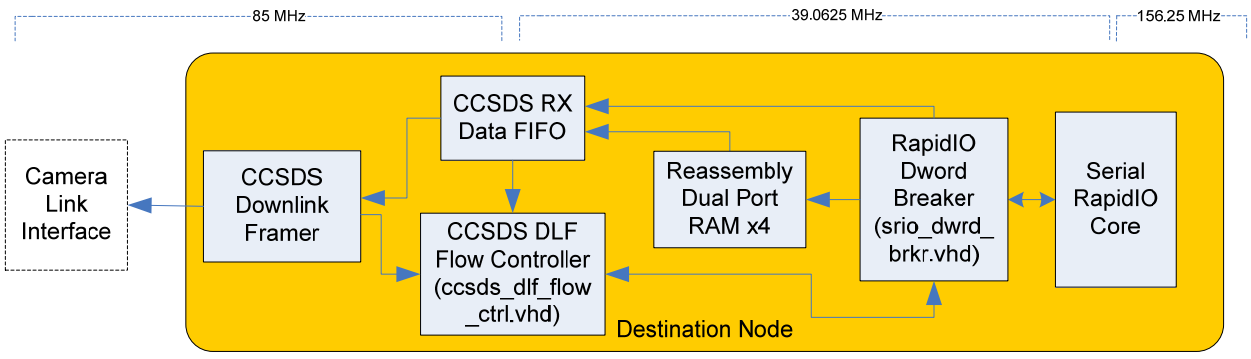

Figure 13. Destination node architecture.

3.3.1 RapidIO Dword Breaker Module (srio_dwrd_brkr.vhd)

The RapidIO Dword Breaker reassembles CCSDS packets that have been broken up into one or more RapidIO message class frames. The current version of this module can reassemble CCSDS packets from up to four CCSDS packet sources simultaneously. This version also only supports up to one outstanding CCSDS packet from any source node at a time. The message class frames that make up each CCSDS packet may be received in any order and may be intermixed with message frames from other source nodes.

As an example, assume in Figure 14 below that "S" refers to a "Source" and "A/B" refers to RapidIO message frames from source nodes A or B. The number following each source identifier is the message segment identifier. A value of '0' refers to the first message frame of the CCSDS packet, a value of '1' refers to the second message frame of the packet, and so on. The figure shows how one possible sequence of message frames could be received from sources A and B if each source was simultaneously sending a CCSDS packet consisting of three RapidIO message class frames to the destination node.

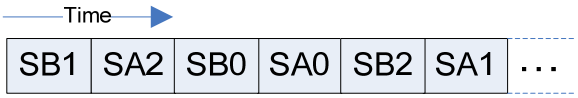

Figure 14. Example flow for incoming RapidIO frames.

The message frame data from each source node is stored in its own re-ordering RAM module (i.e., each of the four RAM is used for a single CCSDS packet from each of the four source nodes) that is indexed using the message frames "msg_seg" field. The RAM block used is based on the source node's device ID field stored in each frame. This means that the maximum allowed number of source nodes in the network is equal to the number of re-ordering RAM modules in the destination design (see Section 3.9 for suggestions on bypassing this limitation).

As each message frame for a given CCSDS packet is received by the module, a corresponding response packet is sent back to the original source of the frame. This response will free the corresponding storage location in the source node's TID Block RAM module. When all message frames for a CCSDS packet have been received, the source node's CCSDS IREQ Generator will be informed that it can send another CCSDS packet.

Any CCSDS packets not aligned on a 64-bit boundary will have the proper number of 16-bit chunks of each RapidIO double-word ignored before being written to the CCSDS packet RX data FIFO. This must be done as the Downlink Framer will not accept any trailing/unused data and will assert an error if it sees any within a packet.

The breaker module also has error checking to ensure that it does not receive duplicate message segments for any CCSDS packet. This error checking is made possible by setting a valid/invalid flag within the re-ordering RAM block as each frame is received. In the current implementation

an error of this nature is unrecoverable and will cause the internal state machine to permanently enter an error state for debugging purposes.[15]

This module also contains a custom 32-bit RapidIO maintenance register (read-only) that can be accessed by the user through RapidIO maintenance read requests from any endpoint in the system. The register allows the user to ascertain the status of the module by providing information regarding the current state of all internal state machines and the number of RapidIO dwords yet to be received for the current CCSDS packet being processed for two out of the four possible sources.

3.3.2 CCSDS Downlink Framer Flow Controller (ccsds_dlf_flow_ctrl.vhd)

The flow controller is necessary to prevent the DLF module from reading partially written packets out of the CCSDS Packet RX Data FIFO. Without this module the DLF might run out of CCSDS packet data before reaching the end of the packet and the packet will be discarded because the DLF cannot support source side stalls.

In order to prevent any CCSDS packets from being discarded by the DLF, the flow controller increments/decrements a counter that represents the number of complete packets stored in the CCSDS Packet RX Data FIFO. The counter is incremented when a complete packet is written to the FIFO and decremented when a complete packet is read out of the FIFO by the DLF.

One critical design attribute that must be taken into consideration when using this IP in any other designs is to ensure that the CCSDS Packet RX Data FIFO is *at least as large* as the largest CCSDS packet that will be sent by the source node. If this rule is not strictly followed, the source and destination node endpoints may reach a deadlock state.[16] The current code revision includes VHDL "assert" statements that ensure that this requirement is met before simulation/implementation.

This module contains a custom 32-bit RapidIO maintenance register (read-only) that can be accessed by the user through RapidIO maintenance read requests from any endpoint in the system. The register allows the user to ascertain the status of the module by providing information regarding the current state of all internal state machines and the number of complete CCSDS packets currently available in the Packet RX FIFO.

3.3.3 CCSDS Downlink Framer Module

The CCSDS DLF module was not designed specifically for this study and has been used before this design. However, it is worth noting in this document as it is part of the destination node's CCSDS packet reception pipeline.

The DLF was used in this study to consume CCSDS packets from the CCSDS RX Data FIFO and then encapsulate them in fixed-length CCSDS frames. The fixed-length frames were required in order to convert the CCSDS data into an acceptable format for later processing by the

[15] This error will never occur provided the source node IP created for this demonstration is used.

[16] Note that this deadlock has nothing to do with the RapidIO or CCSDS protocols themselves; it is merely a design rule that would need to be followed in any packet transfer architecture.

CameraLink device. If there are any errors within any of the CCSDS packets that the module processes, the DLF will assert appropriate errors (e.g., start-of-packet error, end-of-packet error, etc.).

Completed CCSDS frames are subsequently reformatted and sent off chip to another development board, which transmits the CCSDS frames to a desktop computer using the CameraLink protocol. The CCSDS frames provide a synchronization word that is subsequently used to align the data on the ground station. The CameraLink protocol simply provides a mechanism to transport CCSDS frames and packets into the PC using legacy hardware where the user can view CCSDS data and calculate various throughput and data statistics.

3.4 CCSDS Over SRIO Self-Verifying Test Bench

The design and implementation of a complex protocol system such as this requires a well-defined, self-verifying test bench in order to produce good results in a reasonable timeframe. The test bench created for this system verifies proper operation of every hardware component in the system from the first stage in the transmit pipeline on the source node to the last stage in the receive pipeline on the destination node. The test bench was written in Verilog because it was decided that it is more "test bench friendly" than VHDL and would greatly shorten the required implementation time.

The current version of the test bench is limited to verification of transactions on a point-to-point connection between a single source node and a single destination node since Tundra does not provide any structural simulation models for their switches. Consequently, it also cannot verify proper communication between the Tundra switch and any nodes. The test bench must also be run within the ModelSim simulation environment as ModelSim-proprietary library functions are used.

3.4.1 Test Bench Top-Level (cos_to_clink_tb.v)

The top-level module of the test bench provides a framework for the user to make various task calls which can be used to generate traffic, verify traffic flows, and provide end-to-end traffic statistics. Additionally, the top level also initializes all variables at the start of simulation, resets both endpoints, and waits for both endpoints to achieve physical layer synchronization before allowing any traffic to be generated.

In the post-synchronization stage, the test bench will automatically initialize and configure certain maintenance registers within each endpoint to ready them for frame transmission and reception. Only after this final setup stage is complete can the user begin traffic generation.

The user is also allowed to alter various parameters (i.e., using `defines) before beginning the simulation which change the behavior of the test bench at run-time. A few of these parameters are listed below:

- **Debug Level** – Changes the debug output verbosity of the test bench during run-time. Allowed values are 0–3.

- **CCSDS Timeout** – Length of time the test bench should wait for all CCSDS packets to be processed by both the source and destination nodes.

- **Maintenance Timeout** – Length of time the test bench should wait for all Maintenance response packets to be received by the originating node.

- **CRC Storage Memory Size** – Maximum memory size for the CRC verification memory matrix.

- **Source/Destination Device ID** – The node IDs that should be used for the initial endpoint configurations.

- **Enable Maintenance Request Emulator** – If the user does not wish to generate any RapidIO maintenance frames (or instantiate the maintenance frame generator module), but they still wish to verify the functionality of the IREQ Generator Multiplexer, they can use a built-in emulator that will pseudo-randomly toggle the "Request" input to the IGEN Mux component to simulate the transmission of Maintenance frames without actually sending any.

While the simulation is running, the test bench will also run a CCSDS packet CRC checker task in the background to ensure that all CRCs for each CCSDS packet are sent and received properly. The CRC checker uses memory on the local host to store CRC values as they are transmitted and then validate them as they are received by the destination node. The amount of memory space allocated to this function is directly related to the number of "small" CCSDS packets that the user wishes to send. If the user wished to send many (e.g., thousands) of small CCSDS packets then the default memory size will likely need to be increased. The reason for this is because many thousands of small packets may be buffered up on the source node before the first packet is ever received by the destination node, thus causing the CRC memory to overflow because the CRC checker can not clear any validated CRCs. This feature should not be disabled.

3.4.2 Test Bench Task Functions

The current version of the test bench has function libraries available for both the CCSDS packet and maintenance frame generator modules. The user need only include these function libraries (with `include directive) to gain access. The test bench also includes some useful named constants (`defines) for commonly used values that can be used for some function parameters. The tasks are split into two distinct libraries: CCSDS packet generator tasks and maintenance frame generator tasks. The functions in each of these libraries are described below.

3.4.2.1 CCSDS Packet Generator Functions (tasks_ccsds.v)

- **CCSDS_SEQ_GEN(…)** – This task allows the user to generate CCSDS packet sequences using the CCSDS Packet Generator module. If the user starts an infinite packet sequence the infinite sequence can be stopped by calling this task again and loading a finite sequence configuration into the generator module. The user can also set various parameters for configuring the packet generator. These options are described below:
  - Auto Time Generation On/Off – If asserted, enables automatic time generation (sequential 48-bit count). Otherwise, coarse/fine time taken from external data ports.
  - Auto User Data On/Off – If asserted, enables automatic user data generation (sequential 16-bit count). Otherwise, user data taken from external data port.
  - Random Size Payload – If asserted, each packet generated is pseudo-randomly sized. Otherwise, value from "Number of User Words" parameter is used.
  - RapidIO Header Options – Consists of the RapidIO frame header options to be used. These options include the CRF flag, the priority level, the device ID of the target, and the hopcount to the target.
  - Number of Packets – Number of packets to send. If all ones (1s) the generator will send infinite packets.
  - Number of User Words – Number of user words to place in payload (must be within specification). If "Random Size Payload" parameter is enabled this value is ignored.

- **WAIT_CCSDS_SEQ_COMP(…)** – If called, this task will wait for the current CCSDS packet sequence to complete before allowing any more packet generation task calls. If the CCSDS packet sequence does not complete within the user-defined timeout period, the task will assert an error condition. The function parameters are described below:
  - Timeout – How long the task should wait for any remaining sequences to finish.

- **GEN_CCSDS_SEQ_STAT** – If called, this task will generate statistics for all CCSDS sequences that have been sent so far. It will check for any errors and verify that all packet CRCs have been validated by the CRC checker. If any anomalous events occurred during packet transmission or reception the user is informed of what went wrong. This function has no input parameters. An example of the generated statistics output is shown in Figure 15.

```
# CCSDS Over SRIO to Camera Link Solution Statistics:
# ---------------------------------
# ***Maintenance Request Emulator Acesses: [          0]
# ***Maintenance Request Emulator Passes : [          0]
# ***FIFO Overflow/Underflow Errors: [          0]
# ***CCSDS Packets Generated:        [        403]
# ***CCSDS Packets Received by Target: [        403]
# ***CCSDS Packet  Responses Received: [        403]
# ***Valid CCSDS CRCs Received:      [        403]
# ***Invalid CCSDS CRCs Received:    [          0]
# ***Invalid CCSDS Framer Packets Sent: [         24]
# ***Invalid CCSDS Framer Packets Rcvd: [         24]
#
# CCSDS Framer Statistics:
# ------------------------
# ***Start of Packet Errors: [          0]
# ***End   of Packet Errors: [          1]
# ***General  Packet Errors: [          2]
#
# WARNING: The CCSDS Framer module reported that it received invalid pkt formats!
# WARNING: If you did _NOT_ intend to send invalid framer pkt formats then this should be checked!
Otherwise, these warnings may be ignored.
#
# ***All Invalid Framer Packet Formats Were Accounted For***
#
# ********************************
# ***All Packets Were Verified***
# ***  CCSDS SEQUENCE PASSED  ***
# ********************************
```

Figure 15. Statistics output from GEN_CCSDS_SEQ_STAT function.

3.4.2.2 Maintenance Frame Generator Functions (tasks_maint.v)

- **MAINT_SEQ_GEN(…)** – This task allows the user to generate CCSDS packet sequences using the CCSDS packet generator module. If the user starts an infinite packet sequence, the infinite sequence can be stopped by calling this task again and loading a finite sequence configuration into the generator module. The user can also set various parameters for configuring the packet generator. These options are described below:
  - Local Access – If asserted, the maintenance request will be sent to the local endpoint and never be sent across the network.
  - CRF Flag – If asserted, the maintenance frame should be sent as a critical request flow.
  - Priority – Priority level of the maintenance frame. Valid values are (0-2).
  - TType – The transaction type of the maintenance frame (e.g., 0 = Read Request, 1 = Write Request).
  - Data – 32-bit value to be written to the offset register in the case that a write request is being generated.
  - Address – Offset of maintenance register within the endpoint's memory space.
  - Destination ID – Device ID of the endpoint being targeted for the transaction.
  - Hopcount – Hopcount to the endpoint being targeted for the transaction. Ignored if the "Local Access" flag is asserted.
- **WAIT_MAINT_SEQ_COMP(…)** – If called, this task will wait for the current maintenance frame sequence to complete before allowing any more frame generation task calls. If the maintenance sequence does not complete within the user-defined timeout period, the task will assert an error condition. The function parameters are described below:
  - Timeout – How long the task should wait for any remaining sequences to finish.

- **GEN_MAINT_SEQ_STAT** – If called, this task will generate statistics for all maintenance frames that have been sent so far. It will check for any errors, and if any anomalous events occurred during packet transmission or reception the user is informed of what went wrong. This function has no input parameters. An example of the generated statistics output is shown in Figure 16.

```
# Maintenance Packet Statistics:
# ---------------------------------
# ***Maintenance Request Emulator Acesses: [          0]
# ***Maintenance Request Emulator Passes : [          0]
# ***Maint Request  Pkts Generated:        [         12]
# ***Maint Response Pkts Received :        [         12]
# ***Maint Request Pkts Received (Local):  [          4]
# ***Maint Request Pkts Received (Remote): [          8]
#
# ****************************************
# ***All Packets Responses Were Verified***
# ********  MAINT SEQUENCE PASSED  **********
# ****************************************
```

Figure 16. Statistics output from GEN_MAINT_SEQ_STAT function.

3.4.3 Signal Monitors and Signal Spys (signal_<mons/spys>.v)

The signal monitor and spy libraries are what allow the test bench to drive, interface with, and verify the entire hardware design. The monitor library contains most of the process blocks that perform the auto-verification functions. The process blocks in the monitor library should not be altered. The spy library calls on proprietary ModelSim functions that allow the test bench to interface to the hardware and monitor various internal signals without any modifications to the design itself. The spy library was designed in such a way as to allow easy modifications to the library should the user wish to monitor any additional signals.

3.5 Image Generation Module

As part of this research, it was deemed necessary to create a demonstration that would exercise the results of the study and simultaneously display a real-time, visual representation of traffic flow through the test network. To meet this goal it was decided that the transportation of image data across the network would be a realistic application of this design. For example, a Focal Plane Array (FPA) might transmit image data across the RapidIO network, using CCSDS packets, for transmission to a ground station where it could later be analyzed.

3.5.1 Theory of Operation

To represent this process, three source nodes were used to imitate three FPA modules transmitting image data to a DLF. Each FPA module was responsible for transmitting the red, green, or blue color component of every pixel within a complete image. As the image data is received by the DLF module, it is transmitted to a desktop computer using the CameraLink protocol. As the image data is received by the computer, a custom application is used to reconstruct each pixel from the separate red/green/blue components and then display it to the screen.

Because each source node is responsible for one of the three [R,G,B] color components of every pixel in the image, if any one of the source nodes stops transmitting its portion of the image the color of the final image will be distorted as any missing colors will be filled in with zero (0) values. Figure 3 shows the progression of the received image data as each color source node is enabled using the following sequence: red source, green source, blue source.

3.5.2 Fetching the Original Image Data (bmpParse.c, Gen_LCD_Image.java)

Because our design has no digital FPA/camera source from which to capture image data, it was decided that the next best (and simplest) method was to read data directly out of the FPGA's Block RAM. The only issue in this case was how to place image data inside the Block RAM so that it could be used by the hardware.

To accomplish this task, a simple ANSI C bitmap (BMP) file parser was created that generates Xilinx Block RAM Coefficient (COE) files for each red, green, and blue component of each pixel in any input image. The COE files can then be read directly by Xilinx's CoreGen tool to generate a Block RAM instance and initialize all appropriate data values within that memory space. In this experiment, three Block RAM netlist files were created for each color component of the image. Three identical source nodes were then implemented with each node receiving a different color Block RAM.

Before testing the image data in hardware, and to ensure that the COE data files generated by the BMP parser application were valid, another application was written using Java to read in all three COE files and then display the image that should ultimately be received by the destination. This application relies on the Java Swing and AWT libraries to create the image.

3.5.3 Image Generation Hardware (image_gen_bram.vhd)

With the image data stored in the Block RAM it can be read out using a simple finite state machine. The Image Generator module reads image data out of the Block RAM and sends it to the CCSDS packet generator's external user data input ports as necessary. A handshaking mechanism between the two modules ensures that no data is lost. As the image data is received by the packet generator it is inserted into the payload portion of each CCSDS packet being sent.

The image generator must be given the image's original height and width dimensions (in pixels) before being synthesized. If the dimensions are incorrect the image may appear skewed or otherwise distorted when it is recomposed at the final destination.

3.6 Debug and Analysis with RapidFET™

For solving high-level, topological connectivity and traffic-flow issues, a low-level logic analyzer may not be the best or fastest solution for debugging the system. In these instances, a tool created by Fabric Embedded Tools Corporation (FETcorp) [12] called RapidFET Professional was utilized to analyze the network.

RapidFET combines a desktop application user interface with a hardware-based probing device. The RapidFET Probe is attached to the client PC via a 10-Mbit Ethernet interface and is inserted

into the test topology as another endpoint in the system. The probe physically connects to the system using an Infiniband (a.k.a. CX4) cable. In our design, the STx SRDP [7] only had a single CX4 connection available, which we needed to be available for topology studies. To work around this limitation an adapter card was purchased that converts one of the four available AMC ports on the SRDP board into CX4 ports. The CX4-AMC adapter card is available from FETcorp and is shown in Figure 17.

Figure 17. CX4-AMC Adapter Card (courtesy Fabric Embedded Tools Corporation [12]).

The client software communicates with the probe and allows the user to perform various analysis and monitoring tasks including: active/passive topological discovery, link state-of-health monitoring, maintenance register reads/writes on any connected node, routing-table manipulation, and traffic generation. A screenshot of RapidFET being used within a large RapidIO network is shown in Figure 18.

In addition to the aforementioned tasks, the user can also view internal switch state-of-health statistics in the form of real-time, automated graphs on the client-side PC. These graphs are generated by utilization statistics registers and counters available in most RapidIO switch platforms. These registers can be configured with simple maintenance transactions to count specific frame types, packet retries, and multicast events. The user can also choose whether to count inbound, outbound, or bi-directional flows. A screenshot of these utilization graphs is shown in Figure 19.

Aside from the RapidFET Probe device, FETcorp also includes the server source code and libraries that run on the probe with their RapidFET Professional product. This allows the server to be ported to any custom embedded application using the RapidIO protocol. FETcorp has created application notes and instructions on how to port the server code to some designs along with information on creating the necessary “shim” interface to any custom IP.

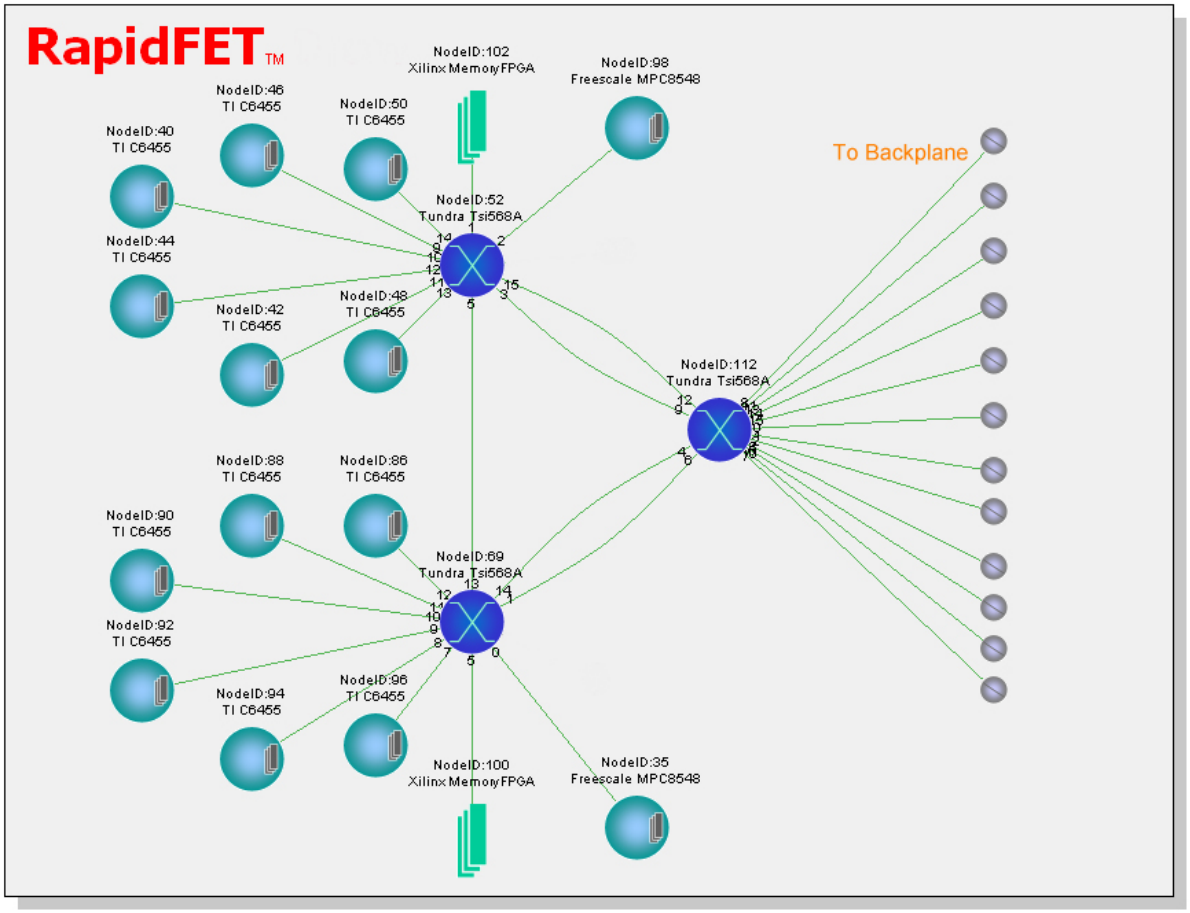

Figure 18. RapidFET Professional and Probe in large RapidIO network.

3.7 Debug and Analysis with the NEX-SRIO

While the RapidFET tool is useful in diagnosing higher-level traffic flow issues, it does not provide the hardware-layer protocol-level view that is required to debug certain design flaws. For these issues, a different tool, created by Nexus Technology, Inc. called the NEX-SRIO Protocol Analyzer, was used. The probe used by this device sits physically inline with the traffic flow between two nodes. The probe is then attached to a back-end pre-processor box which, in turn, attaches to a TLA Tektronix Logic Analyzer [13]. A block diagram of this topology is shown in Figure 20.

The paths defined in Figure 20 are defined below (directly adapted from Nexus website):

- **Path A** - Connection between the system-under-test and the probe. This can be a NEX-MIDBUS probe or two to eight NEX-SERIALPROBE probes (two for a single direction x1 link and up to eight for a bi-directional x4 link). The midbus probe was used in this design as it is much easier to work with than the serial probes. The serial probes require precise soldering while the midbus probe utilizes a simple screw-down mechanism. The CX4-SMA adapter board required to use the Nexus midbus probe is shown in Figure 21.

Figure 19. RapidFET Utilization Graphs.

- **Path B** – The probe(s) are connected to the pre-processor on a channel-by-channel basis.
- **Path C** – The pre-processor uses four or six P6860 probes to send the de-serialized data to the logic analyzer.
- **Path D** – The P6860 probes connect to two modules on the logic analyzer. The logic analyzer then triggers, stores and disassembles, and displays the data for the user.
- **Path E** – The USB connection is used to transfer setup information to and from the pre-processor using proprietary software included with the analyzer hardware.

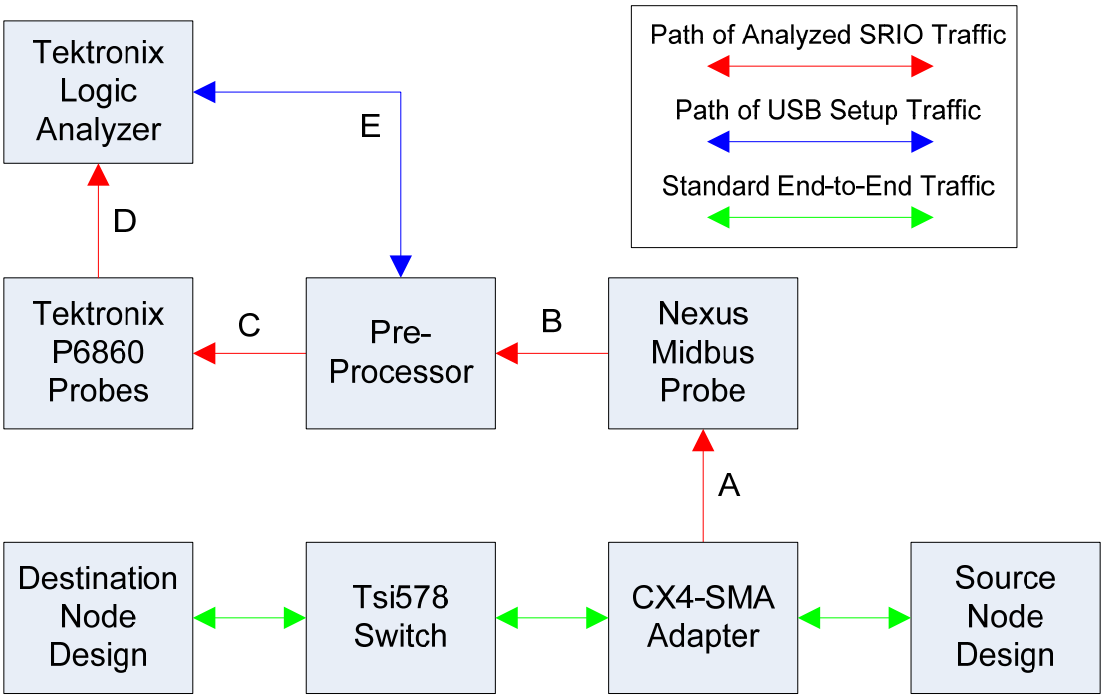

Figure 20. Nexus SRIO Protocol Analyzer connection topology (adapted from [14]).

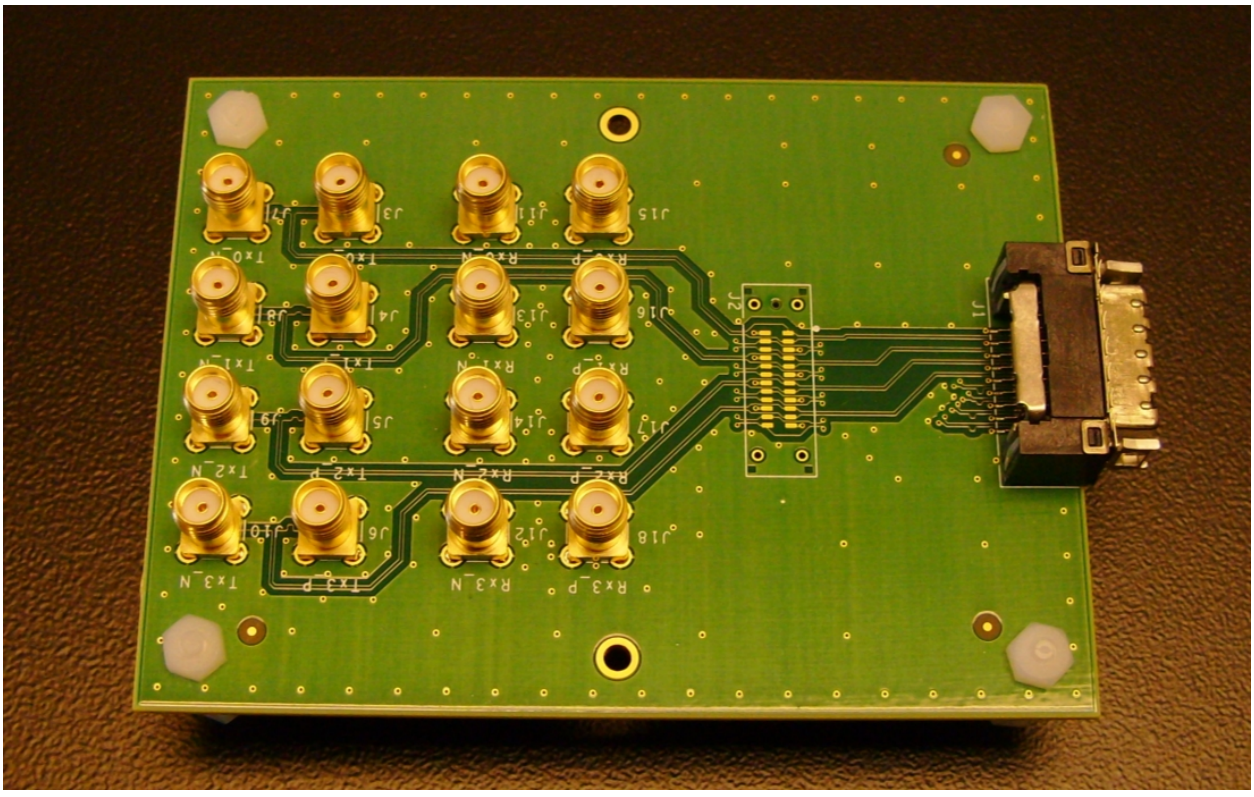

Figure 21. CX4-SMA adapter board (courtesy Fabric Embedded Tools Corporation [12]).

The proprietary software that must be installed on the logic analyzer allows the user to set up the pre-processor box for viewing bi-directional SRIO packet flow across the adapter card up to the maximum allowable RapidIO link rate of 4x3.125 Gbps. A screenshot of the software's SRIO frame disassembly view is shown in Figure 22. This view breaks each SRIO frame into hierarchical layers from packet and control symbols down to the bit level. It can also group SRIO request frames with their associated response frames, which allows the user to easily navigate through the captured traffic.

It is important to note that the Tektronix logic analyzer must be equipped with a minimum 450-MHz state speed acquisition module (TLA7xx2/3/4) with two to three P6860 probes for a single SRIO datapath. These requirements must be doubled for two SRIO data paths [14].

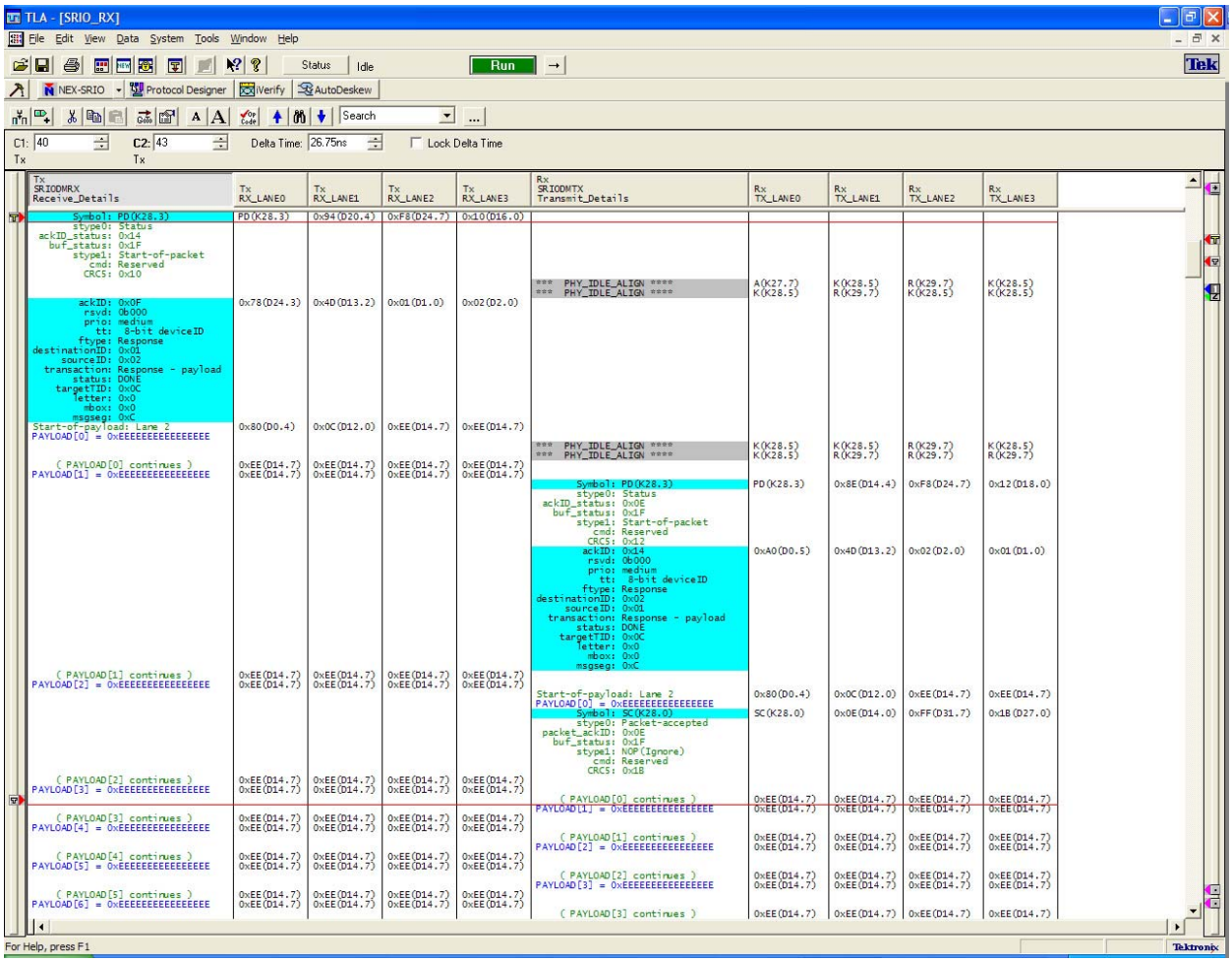

Figure 22. NEX-SRIO packet disassembly software.

3.8 Debug and Analysis Setup with STx SRDP

In this design, both the RapidFET Professional tool and the NEX-SRIO module were utilized for debug and analysis. A high-level block diagram of this setup is shown in Figure 23. The system shown requires (at minimum) a single CX4-to-SMA adapter and a single CX4-to-AMC adapter from FETcorp [12]. If another SMA interface is desired, for possibly attaching more endpoints, an SMA-to-AMC adapter card can also be purchased from STx [7].

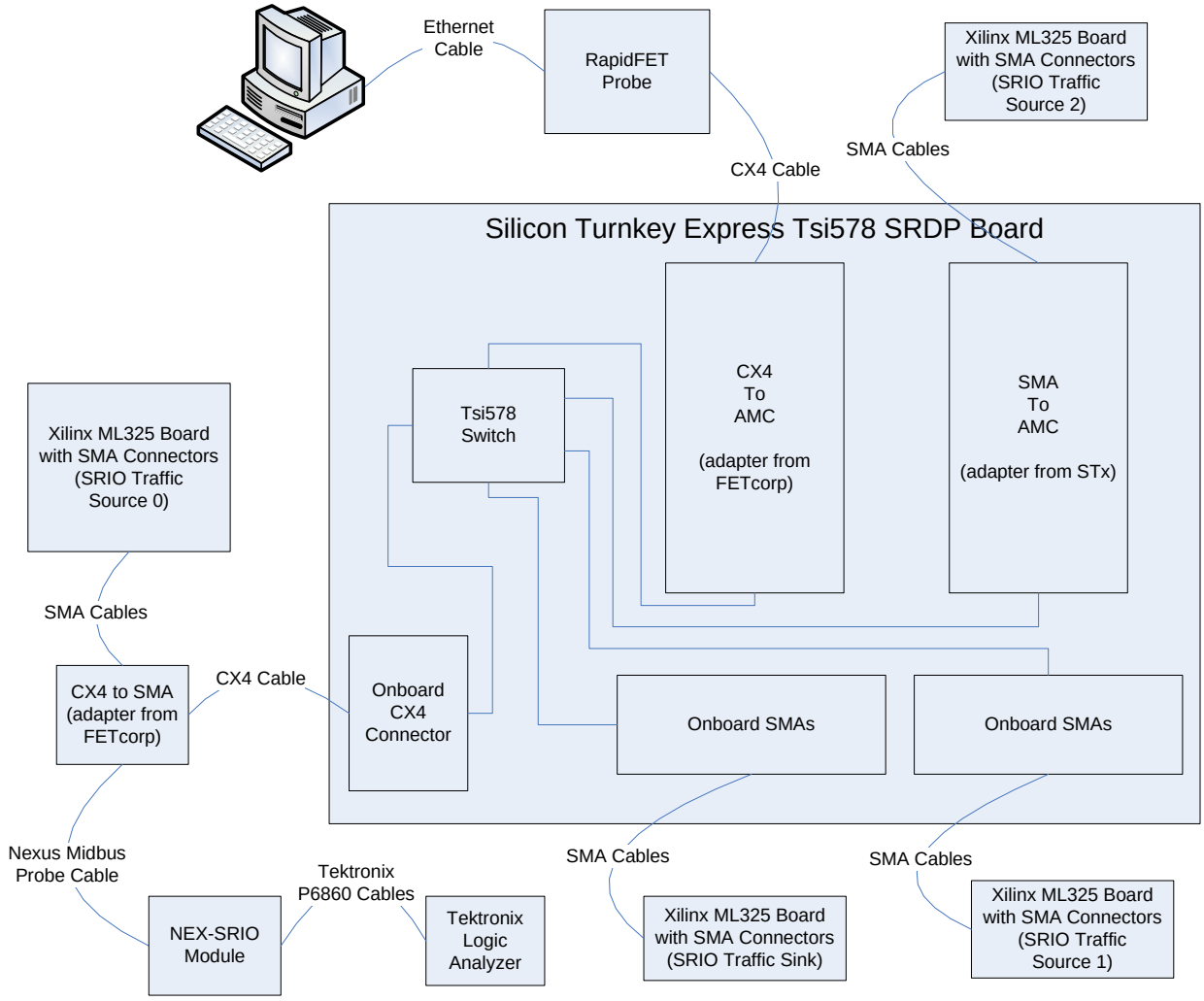

Figure 23. Debug and analysis system setup.

3.9 Future Work with RapidIO

Many improvements will be made to the source and destination node designs in future revisions. As the design and system topology becomes increasingly complex, many of the current limitations will need to be overcome. This section discusses a few of those limitations and how solutions for them might be implemented.

- **Maximum Allowable Source Nodes** – A maximum of four source nodes is allowed in the current design. This is by far the most notable limitation in the design. To correct this issue the destination node design would need to be modified to treat each of the four (or more) re-ordering RAM components as "the number of available CCSDS packet buffers" rather than assigning each RAM to a specific source node. This upgrade will require some form of handshaking protocol in which the source node first checks if there are any free buffers available before beginning transmission of a CCSDS packet. This handshaking mechanism will add support for an unlimited number of source nodes.

- **Multiple Source Packets Per Receive Buffer** – The current implementation of the destination node only allows a single CCSDS packet to be stored in each re-ordering RAM buffer regardless of its size. An upgrade would allow multiple CCSDS packets to be stored in a single RAM block. This design change would require an additional context memory space to hold a linked list of the beginning and end of each packet with the RAM space, however, it would remove the requirement of having multiple RAM blocks in the destination node. This would also require a slightly more complex handshaking mechanism in order to allow a source node to check for available buffer space at the destination node before transmitting a CCSDS packet. Similar to the previous bullet, this upgrade would also allow for an unlimited number of source nodes in the network.

- **Automated Discovery** – The LUT population in the Tundra switch is manually performed in software via the available RapidIO API functions. The software can be upgraded to automatically discover endpoints using the same API functions by integrating those functions into the RapidIO standard discovery algorithm. Annex 1 of the RapidIO specification Version 1.3 includes pseudocode for the algorithm [8].

- **Interrupted Driven Interface** – The CTSS IP core is currently polled for maintenance response frames and error events using the tight-loop polling method. Future revisions of the core will be interrupt-driven. This can be achieved by simply adding an interrupt controller to the design and attaching it to the PowerPC's bus architecture.

- **Quality of Service** – The QoS feature of RapidIO was not utilized in this demonstration; however, it would not be difficult to add this functionality in future designs. The most logical way to set the priority flags would be to assign a priority to a particular CCSDS packet and thus keep that priority constant across all RapidIO frames for that CCSDS packet. Subsequently, on the destination node, the priority of each CCSDS packet could be used to store the packet in a particular packet FIFO. This would allow for the creation of a hardware-based packet scheduler that could choose packets of certain priority for forwarding on to the DLF.

- **4x Links** – The current testing topology only utilizes single-channel RapidIO links. By regenerating the Xilinx physical layer core, a 4x rate could be achieved with a few minor design changes and four times as many SMA cables.

- **PowerPC 440** – If the current design was transition to a Xilinx Virtex-5 FPGA, the software-level implementation of the source node design could take advantage of the new PowerPC 440 processor. The 440 boasts a seven-stage pipeline (405 has only five-stage) and out-of-order execution. These two enhanced features would significantly increase software speed and performance as the instruction and data memories could then be placed in separate Block RAMs on two different PLBs.

- **MicroBlaze** – The current software layer drivers could be easily ported to a soft-core processor (e.g., Xilinx MicroBlaze) rather than the hard-core PowerPC. The choice of soft-core processor would be based on [15].

4. SPACEWIRE DEMONSTRATION

The SpaceWire protocol is a serial data protocol developed primarily by the European Space Agency. SpaceWire is a bi-directional, full-duplex serial protocol for use in point-to-point applications. SpaceWire is currently in use in a number of flight systems to provide a high-speed data infrastructure between sensors, processing elements, memory units, telemetry subsystems, and other space instruments [3]. As SpaceWire is already utilized in many space projects today, its feasibility for flight systems has already been proven, making it a promising candidate for integration into a node-based system.

SpaceWire has a relatively comprehensive protocol specification, specifying requirements from cables and connectors through character transmission up to network packet transmission and error handling [3]. The important characteristics of the SpaceWire protocol will be outlined; however, it is recommended to refer to the specification for complete details as necessary.

The physical layer of SpaceWire provides requirements for the physical medium upon which SpaceWire signals are transmitted. SpaceWire was specifically developed to meet the electromagnetic characteristic requirements of typical spacecraft. SpaceWire cables are four-pair twisted-pair cables with individually shielded pairs, plus an overall shield, terminated with 9-pin micro-miniature D-type connectors. Cable length is specified up to 10 meters at the maximum SpaceWire data rate of 400 Mbps, and cable weight must be kept below 80 grams per meter. SpaceWire also may be transmitted on printed circuit board (PCB) traces with 100-Ω differential impedance [3].

SpaceWire signaling utilizes low-voltage differential signaling (LVDS) as specified by ANSI TIA/EIA-644. The use of differential signaling provides a level of noise immunity and the current-driven nature of LVDS ensures a low and consistent power consumption. The specification dictates that SpaceWire run between 2 Mbps and 400 Mbps, although some commercial hardware claims to run SpaceWire at speeds up to 625 Mbps. It should be noted that the maximum speed of SpaceWire may limit the capabilities of future systems requiring higher bandwidths. These higher bandwidth systems may wish to consider Serial RapidIO™ as a more appropriate option for their applications [2]. Data is encoded using data-strobe encoding, which requires two LVDS pairs of wires per direction; thus, a complete SpaceWire link between two nodes requires a total of four differential pairs (eight wires or traces).

SpaceWire utilizes ten bits of information to transmit eight bits of data; however, unlike other protocols, the data is not encoded (such as with 8B10B encoding used by many other protocols). Rather, two bits are prepended onto each data byte to provide parity and an end-of-packet flag for each data byte. The parity bit provides for single error detection and the availability of an end-of-packet flag on every data byte allows for an arbitrary packet length.

In a routed SpaceWire network, data packets are transmitted through the network by providing a short header to the router indicating the data's intended destination, followed immediately by the data payload. The header information is one or more bytes used to identify the destination or destination path for the data payload. In a logical addressing mode, a single byte with the logical

ID of the destination node is provided. As this packet traverses the network, routing tables are used to identify and transmit the packet out the appropriate port to the next hop. For example, a packet [40 <DATA>] would transmit the [<DATA>] payload to node with logical ID 40. In a direct addressing mode, the output port numbers to reach the destination are explicitly specified. For example, the packet [04 02 03 <DATA>] would first be transmitted through port 4 of the first router, then port 2 of the second router, then port 3 of the third router. A third addressing scheme, regional addressing, is not in widespread use and will not be discussed here.

Regarding node addressing, addresses 1 through 31 are dedicated to physical ports on the local SpaceWire router. Thus, when using direct addressing, port numbers must stay below 32. Addresses 32 through 254 are logical addresses and are typically assigned to routers or endpoints. Address 255, although available for use as a logical address, is typically reserved for future expansion. There is no theoretical limit on the number of devices that may be present in a SpaceWire network; an addressing technique known as regional addressing may be used to expand the size of a SpaceWire network to any size.

One important characteristic to understand about SpaceWire networks is their use of wormhole switching technology. Wormhole-switched networks operate by reading the data packet header immediately upon receipt of the first data byte of the packet. Based on this first header byte, the packet is immediately forwarded out the appropriate output port as the remainder of the packet is received. However, in the event that the output port is busy, the partially received packet will stall until the output port is free. In addition to the current router being affected by the stall, any routers behind the stalled router currently switching the stalled packet will also stall, causing a temporary pause in the affected ports of the routers switching that packet until the output port frees. This is typically not a problem, unless there are multiple stalls that result in a circular wait in the network. This condition is referred to as “deadlock” and must be carefully avoided as much as possible [16].

4.1 SpaceWire IP and Hardware Selection

A SpaceWire IP core from NASA’s Goddard Space Flight Center (GSFC) was utilized for the SpaceWire implementation. The GSFC IP core provided models for both SpaceWire point-to-point link and network router models implemented in the VHDL language. These cores have been used in the past by other organizations within SNL for SpaceWire implementations and even by outside commercial companies for production of SpaceWire networking components. Thus, these cores have been readily used in real-word environments and have proven their ability to function and adhere to the SpaceWire standard. The discussion here forward will focus on the GSFC router IP core.

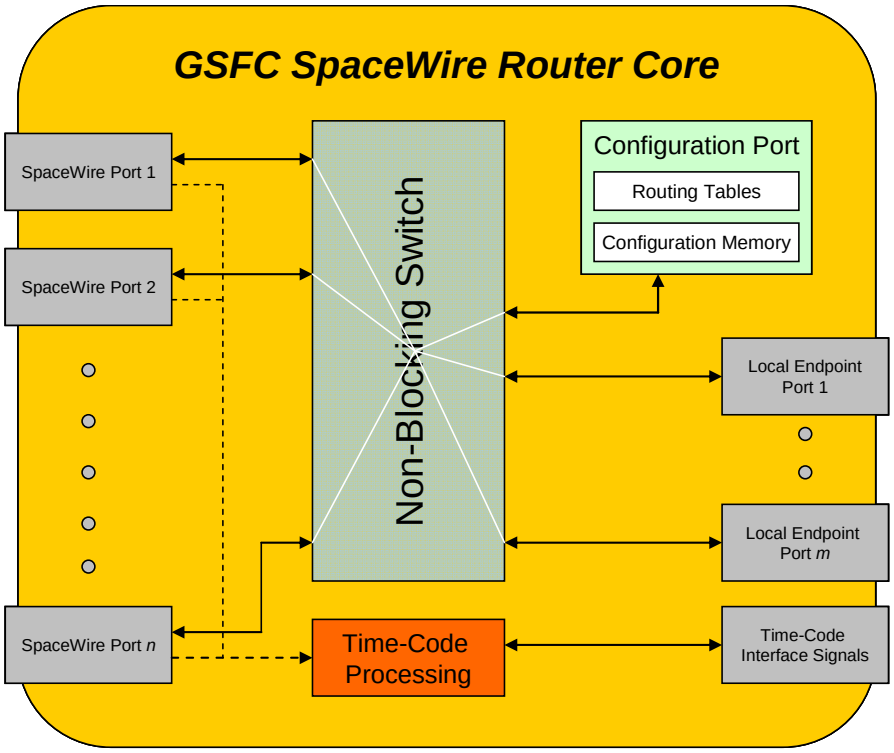

Figure 24. GSFC SpaceWire router IP core block diagram.

The SpaceWire router core, shown in Figure 24, is a configurable router core that provides a non-blocking network switch with a configurable number of ports. The ports in a SpaceWire switch may interface to one of two functions: an external SpaceWire point-to-point link, or a local link for connection to an endpoint (referred to as a "local" link or port in this document). The only restriction is that the number of external SpaceWire ports plus the number of local ports may not exceed 31. Both SpaceWire and local ports are connected to a non-blocking switch, which allows for communication between any combinations of port pairs simultaneously, provided no two sources are attempting to use the same output port. Latency through the switch from port to port is affected by a number of factors, but typically is approximately 30 clock cycles. This includes delays for routing lookups, port arbitration, and other switch functions.

The router core is equipped with several attractive features. Remember that SpaceWire utilizes wormhole switching, which makes it particularly susceptible to deadlock. To maintain traffic flow through the network, the routing switch has the ability to terminate transmission of any packet stalled for some configurable length of time. This ensures that deadlocked packets will not stall the network indefinitely. Furthermore, the router has the ability to automatically fail any hardware link that demonstrates the inability to maintain a reliable link. This prevents use of a faulty or intermittent link for communications. The router incorporates a packet duplication feature, which transmits a network packet out two different (configurable) ports simultaneously, or permits configuration of a secondary port to utilize if the primary port is busy. This permits networks to utilize escape paths (see [16] for details) to maintain a high level of reliability and maximize usable bandwidth in the network. Lastly, the router incorporates a bandwidth throttling capability to reduce the speed on links when they are not actively utilized in order to

save power by reducing dynamic switching in the router logic. Note that not all of these features are incorporated into this hardware demonstration; however, these features are available for use in future SpaceWire implementations.

The router is configurable via packets written to the router configuration port. Configuration packets may originate from local ports or may be received from any external SpaceWire link. These configuration packets allow reading and writing to registers within the configuration memory. These registers hold the routing table information, packet and error counters, port speed and configuration, time code configuration, and a variety of other option configurations and status words.

The IP provided by GSFC was implemented on a number of platforms. Using the Virtex-II Pro based platforms, the GSFC router IP has been tested at speeds up to 170 MHz and could probably run at approximately 200 MHz without significant effort. The SpaceWire core logic runs at one-fourth the line rate, thus the router logic would run at 50 MHz. Running faster than 200 MHz would likely require some development effort to improve portions of the core that impede higher speed operations.

Primary development for the SpaceWire hardware demonstration was performed on a Xilinx ML325 development board as described by the hardware demonstration overview. The ML325 is a prototyping platform built with a Xilinx Virtex-II Pro FPGA (XC2VP70). Other platforms used to evaluate the SpaceWire IP included a Xilinx ML523 development board (populated with a Xilinx Virtex-5 LX110T) and a board created by SEAKR Engineering comprised of two Virtex-II Pro and one Virtex-4 LX FPGAs.

Other hardware utilized by this segment of the project includes two SpaceWire peripheral component interconnect (PCI) cards (to provide SpaceWire connectivity to a PC) and the custom FIB transceiver board described earlier in the hardware demonstration overview. The PCI cards enabled communications between a PC and the SpaceWire network, and were used primarily to verify routing, read or write configurations, or to inject data into the network for testing purposes. The FIB board was equipped with a Virtex-II (XC2V3000) FPGA, four high-speed serial SERDES made by Texas Instruments (TI TLK2501), and three ChannelLink SERDES (National Instruments DS90CR287). The FIB board accepts four high-speed serial data inputs received via coaxial SMA cables, each running at 1.7 Gbps with 8B10B encoding. This provides data at an effective rate of 1.36 Gbps per channel. The data from these four channels is logically bonded and retransmitted using a protocol called CameraLink. CameraLink is a high-speed data protocol implemented with National Instruments ChannelLink SERDES on the FIB board. The ground station PC would receive this CameraLink data via a CameraLink PCI-X interface card at an effective rate of 5.44 Gbps.

4.2 SpaceWire Implementation

Each SpaceWire router instantiation was configured with four external SpaceWire ports and three local ports for communication to endpoints. To save prototype hardware, two independent nodes were instantiated within one FPGA. Although these two nodes are physically located on the same chip, they are completely independent and must be cabled together to communicate, just as if they were on separate boards. This is shown in Figure 25.

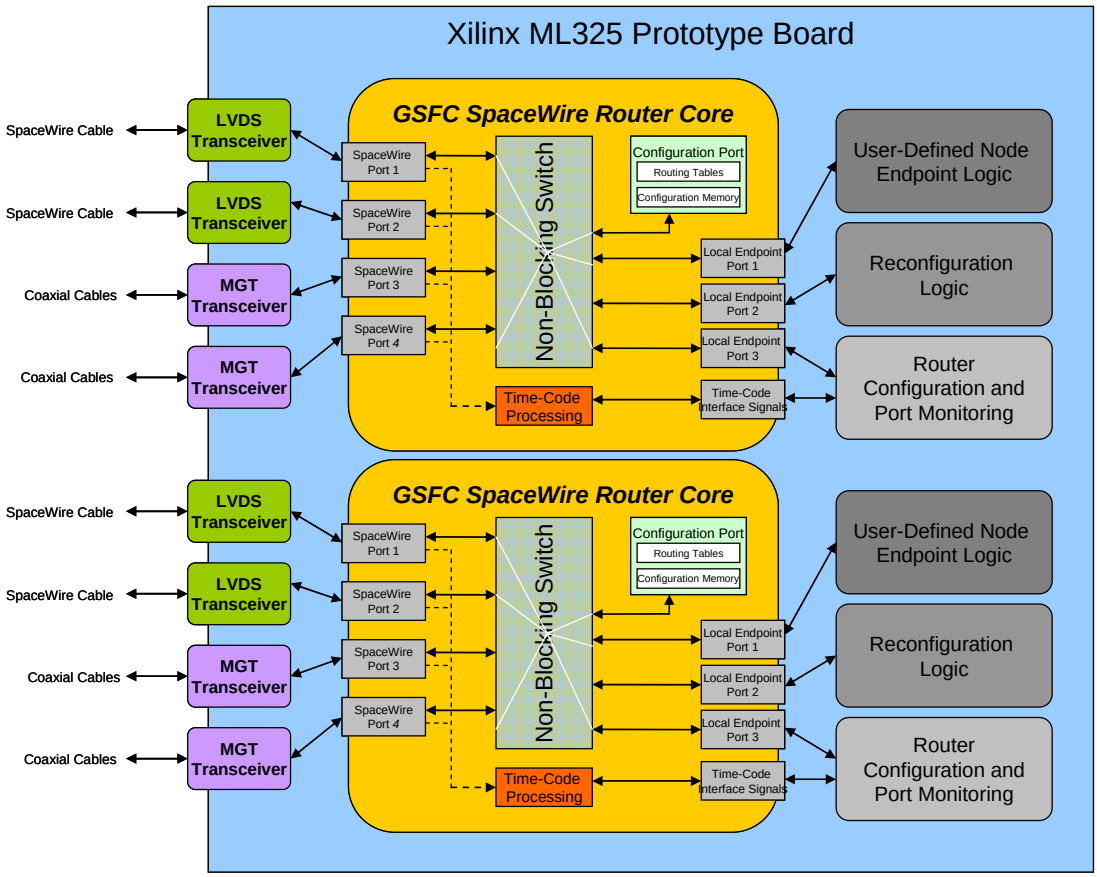

Figure 25. Two independent node instantiations on one ML325 board.

Each external SpaceWire port is attached to one of two HDL modules: an LVDS transceiver or a multi-gigabit transceiver (MGT) transceiver. The LVDS transceiver takes the data and strobe values generated by the GSFC SpaceWire IP core and instantiates the proper OBUFDS differential drivers to drive these signals differentially off-chip. Additionally, IBUFDS differential receivers are instantiated to translate received differential data and strobe values into discreet signals provided to the SpaceWire input for that port in the GSFC router IP.

Due to the lack of differential pin availability on our ML325 demo boards, as well as the lack of proper SpaceWire harnesses and cabling, a second scheme to transmit SpaceWire signals between boards using MGTs was developed. This allowed us to use highly available standard coaxial cabling with SMA connectors as the physical medium for transmission of our SpaceWire signals. Also, this had the added benefit of incorporating the same physical medium as the Serial RapidIO™ demonstration, essentially creating a common physical layer for both protocols.

To properly transmit SpaceWire signals over an MGT, the bit clock feeding the SpaceWire IP core is also used to drive the MGT. This allows us to sample the data and strobe outputs of the SpaceWire IP core at the same frequency as the bit clock (eliminating the need to sample at the Nyquist frequency). These data and strobe values are encapsulated into a data word and

appended to a synchronization byte. On the receiver side, the recovered data clock is used to drive the received data and strobe values into the SpaceWire IP core. By making these signals synchronous with the recovered data clock from the MGT data stream, this eliminates any need for clock correction that may arise from slight oscillator variations between boards.

As mentioned earlier, there are three local ports instantiated per SpaceWire router. One local port is provided for communications to and from endpoint logic. The second local port provides router configuration and port monitoring capabilities. The third local port is connected to reconfiguration logic, which may be used to reconfigure FPGAs (further described in Section 4.3).

The first local port provides network access to endpoint logic. This endpoint logic will be the reprogrammable portion of the node. This will involve study of the partial reconfiguration capabilities of Xilinx FPGAs, which has been tasked through a follow-on LDRD beginning in FY 2009. Until then, the logic and the node endpoint capabilities will remain in a single joint design that remains static while powered until the entire node (both endpoint and router) is reprogrammed.

The second local port provides rudimentary router configuration and port monitoring (RCPM) capabilities. The responsibilities of the logic connected to this port are twofold. First, this endpoint module is responsible for sending the proper configuration packets to the router logic to properly configure the SpaceWire ports upon power-up or reset. This includes the proper setup of static routes to utilize logical addressing for communications between nodes. Second, this module monitors network state in its direct vicinity. It will monitor the link status of the SpaceWire router and notify a remote host (usually a command node) in the event of a link failure or when a link is reestablished.

Previous versions of this RCPM endpoint would also perform network discovery functions and automatic routing table generation. This was performed by querying the remote end of each SpaceWire link to see if another router was present. As remote routers were discovered, the local endpoint would generate routes to these neighbors. It would then periodically share a copy of the entire local routing table with its neighbors. As RCPM endpoints received copies of their neighbor's routing tables, they were able to expand their own routing tables and derive routes to other nodes that were not directly adjacent to them. This scheme would continue until the entire network was discovered and routes were generated to all known nodes. This functionality, however, was temporarily removed as the implementation of the network discovery feature uncovered a bug in the GSFC router IP logic that would cause the router to stop responding. Efforts to modify the core to fix this bug were recently successful; however, the network discovery feature has not yet been reinstated.

The utilization results for a single-node SpaceWire router and endpoints are shown in Table 6 and Table 7. Note that these numbers are for single-node designs only, not the dual-router implementation shown in Figure 25. As shown in the tables, logic utilization is comparable to the RapidIO node designs for source and destination nodes, with the exception of global clocking (GCLK) resources. This is due to the fact that each SpaceWire port requires a global clock net to propagate the recovered clock for that SpaceWire link. Furthermore, each SpaceWire MGT transceiver requires an additional global clock net for the MGT recovered clock. For the actual router core, three global clock nets are used to provide clocks to the router instantiation. These three clocks run at the SpaceWire line rate (156.25 MHz), the SpaceWire router core logic rate (1/4 of the line rate, or 39.0625 MHz), and an endpoint or user-defined logic rate (85 MHz, selected to match the oscillator on the FIB test board to facilitate transmission of data to the spacecraft).

Although these numbers are essentially on par with the RapidIO source node design, it is important to remember that this implementation includes both the network router and source node endpoint logic. Thus, this design includes switching and routing capability whereas RapidIO requires use of an external switch, as no RapidIO switch IP is currently available.

Table 6. Device Utilization Statistics for SpaceWire Single Node Sensor Interface (source node) Design on Virtex-II Pro 70.

DCMs	1 out of 8	12%
Block RAMs	38 out of 328	11%
Flip-Flops	7369 out of 66176	11%
4-input LUTs	11815 out of 66176	17%
GCLKs	9 out of 16	56%

Table 7. Device Utilization Statistics for SpaceWire Single Node Downlink (destination node) Design on Virtex-II Pro 70.

DCMs	1 out of 8	12%
Block RAMs	36 out of 328	10%
Flip-Flops	7888 out of 66176	11%
4-input LUTs	12004 out of 66176	18%
GCLKs	8 out of 16	50%

Figure 26 illustrates the hardware utilized for the SpaceWire hardware demonstration and its connectivity.

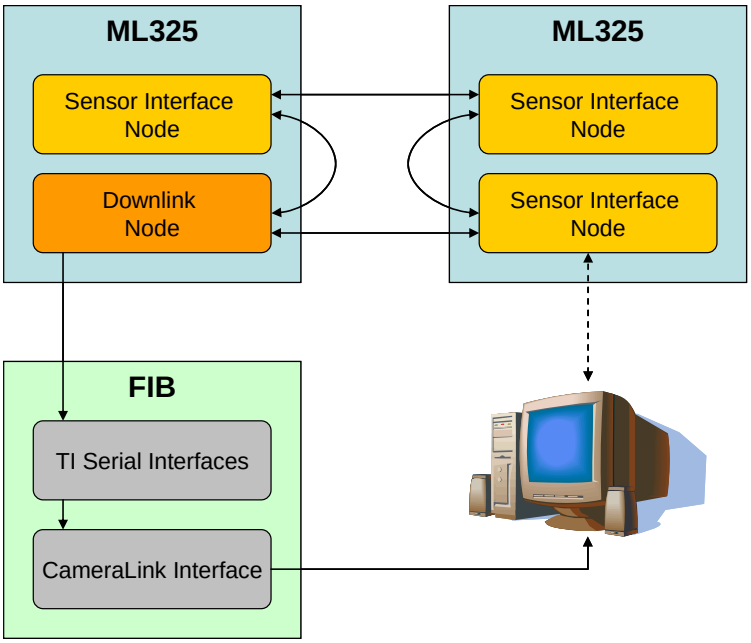

Figure 26. SpaceWire hardware demonstration layout.

The SpaceWire image demonstration successfully demonstrates the ability to route data from three separate sensor interface nodes to one downlink node, which then sends the data to a ground station. This model programs each sensor interface node's endpoint logic with a CCSDS packet generator, which generates CCSDS packets containing one constituent color component of an image. When the packets from the three sources arrive at the downlink node, they are sent to the ground, where the image is reconstructed from the three data sources.

The downlink endpoint is programmed with a CCSDS framer that accepts incoming CCSDS packet data and encapsulates those packets into CCSDS frames. Once the data is in CCSDS frames, it is transmitted to the ground station, where it is stripped of framing data and packet headers. The remaining data is then used to recompose the image. A node failure, simulated by disconnecting the cables to one node, results in a loss of that color component and results in an image with a distorted colormap.

4.3 Remote Configuration over SpaceWire

One key feature of the NBA is the ability to dynamically reprogram endpoint logic in flight to provide different node functions within the network. In addition to initial power-on programming duties, this provides a failover capability to mitigate in-flight failures. A system that experiences a node failure in flight may reprogram one of the spare nodes on the network to replace the function of the failed node, thus keeping the system fully operational.

A special configuration host interface node was created on a Xilinx ML523 development board with a Virtex-5 LX110T FPGA to serve as the configuration host and bitfile source. This board, in addition to having a typical router and endpoint instantiation, also included a soft-core processor and a non-volatile flash memory in the form of a CompactFlash (CF) card. The device

utilization for this implementation is shown in Table 8. As future hardware will most likely be targeting a Virtex-5 FX130T FPGA, the utilization numbers for that architecture are shown in Table 9.

Table 8. Device Utilization Statistics for SpaceWire Single Node Design on Virtex-5 LX110T.

DCMs	1 out of 12	8%
Block RAMs	64 out of 148	43%
Flip-Flops	8483 out of 69120	12%
6-input LUTs	9736 out of 69120	14%

Table 9. Device Utilization Statistics for SpaceWire Single Node Design on Virtex-5 FX130T.

DCMs	1 out of 12	8%
Block RAMs	64 out of 148	43%
Flip-Flops	8485 out of 81920	10%
6-input LUTs	9720 out of 81920	11%

A MicroBlaze™ was instantiated in the Virtex-5 FPGA to serve as the soft-core processor element. The MicroBlaze™ provided both a user interface for bitfile selection as well as the control logic for transmitting the bitfile via SpaceWire. A custom MicroBlaze™ peripheral was created to interface the processor to the SpaceWire router. This allowed the MicroBlaze™ to interface to the SpaceWire router as an endpoint, enabling the processor to send and receive SpaceWire packets via the network. This implementation in the ML 523 board is shown in Figure 27.

To properly demonstrate the remote configuration of a FPGA, the ability to drive and read special configuration pins on the FPGA is required. These configuration pins comprise the SelectMAP programming interface on Xilinx FPGAs. Detailed information regarding this configuration method is available in [17].

Currently the only development board available to this project capable of serving as a target for remote configuration is a board developed by SEAKR Engineering, Inc. This board was designed with two Virtex-II Pro FPGAs and a Virtex-4 FPGA attached via a daughter-card. One of the Virtex-II Pro FPGAs has direct access to the SelectMAP pins of the Virtex-4 FPGA, making this board an ideal candidate to demonstrate remote reconfiguration over SpaceWire.

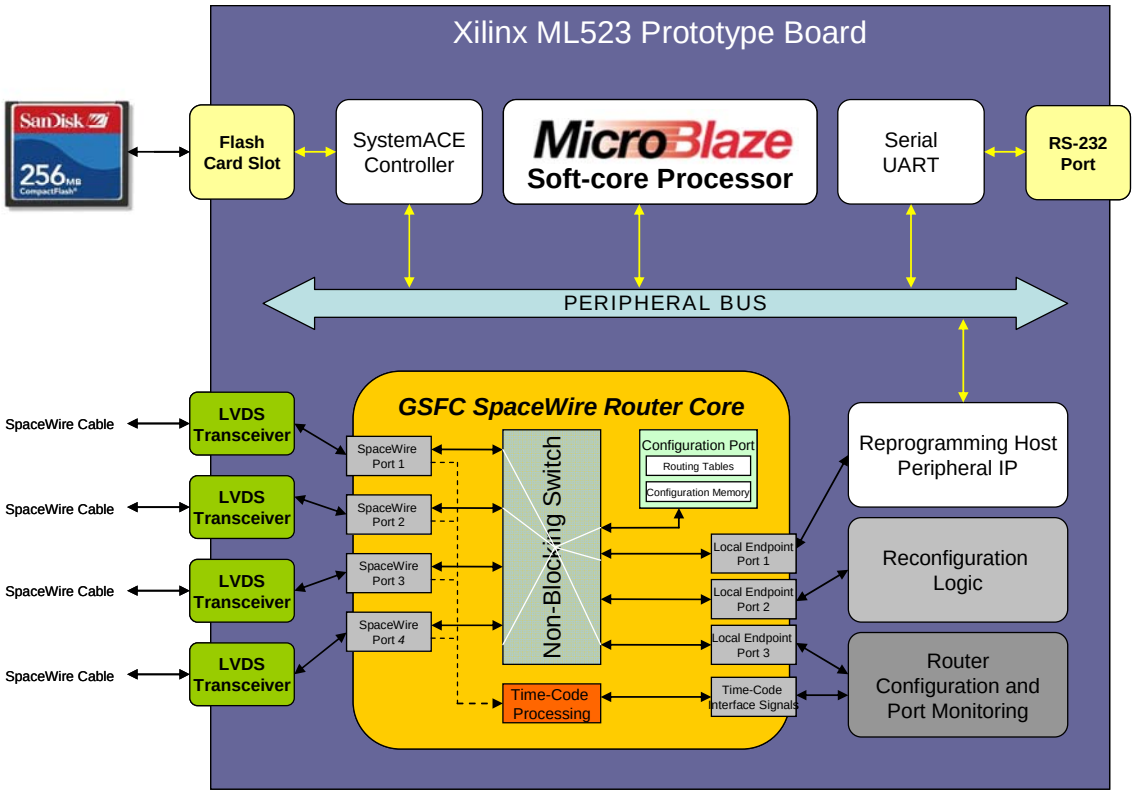

Figure 27. Configuration host (ML523) block diagram.

For the configuration targets, an endpoint module was developed to receive bitfiles remotely via SpaceWire and to drive a SelectMAP configuration interface appropriately. This module is connected to the third local port instantiated on each SpaceWire router. This module receives bitfiles from the ML523 board serving as the configuration host and drives SelectMAP interface pins. On the SEAKR board, these SelectMAP interface pins program the Virtex-4 FPGA. Other boards (such as ML325s) in the network also contain this module; however, their SelectMAP interfaces are not connected, and sending a bitfile to these boards will result in no configuration change. Figure 28 shows the implementation of these functions on the SEAKR board. Figure 29 shows the higher-level architecture for the reconfiguration demonstration.

Figure 30 is an example of the configuration interface, which is used to input a filename, select source and destination node numbers, and begin programming over SpaceWire. The filename allows the user to select a bitfile stored on the CF card. Destination node number specifies the target node to program. Source node must be set if status information regarding the success or failure of programming is desired.

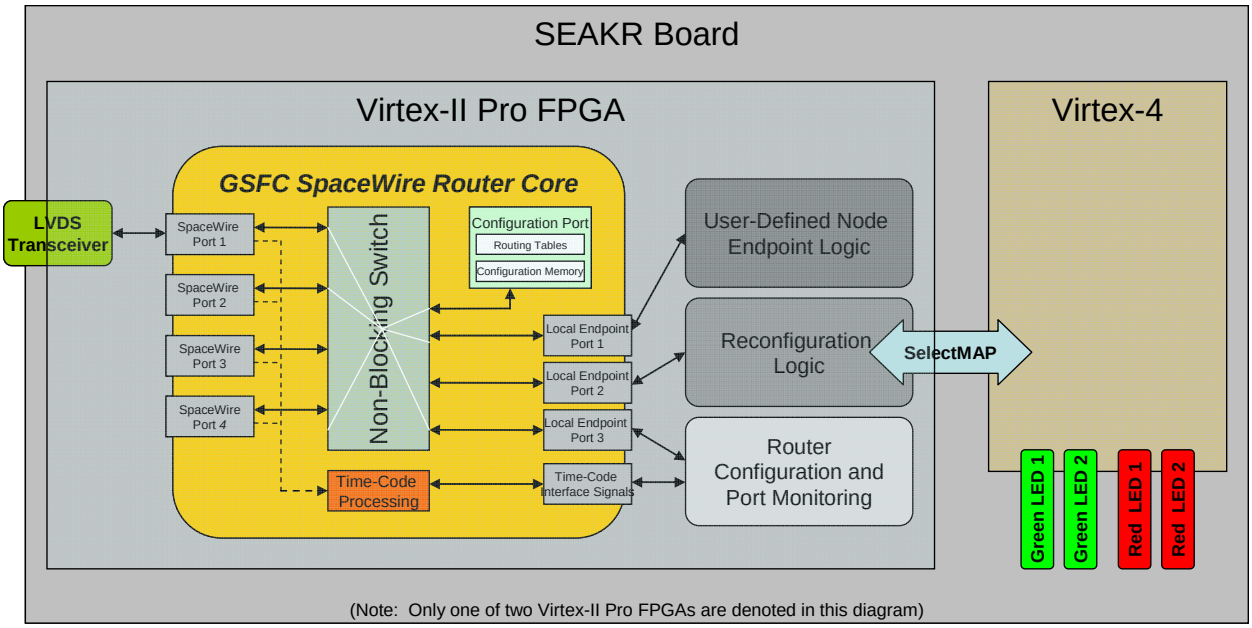

Figure 28. Configuration target (SEAKR) block diagram.

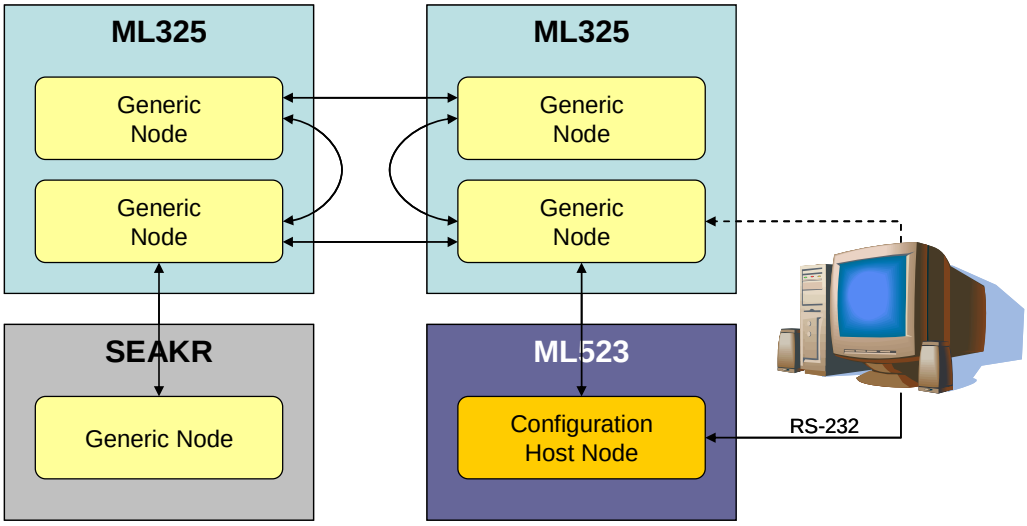

Figure 29. System block diagram for reconfiguration demonstration.

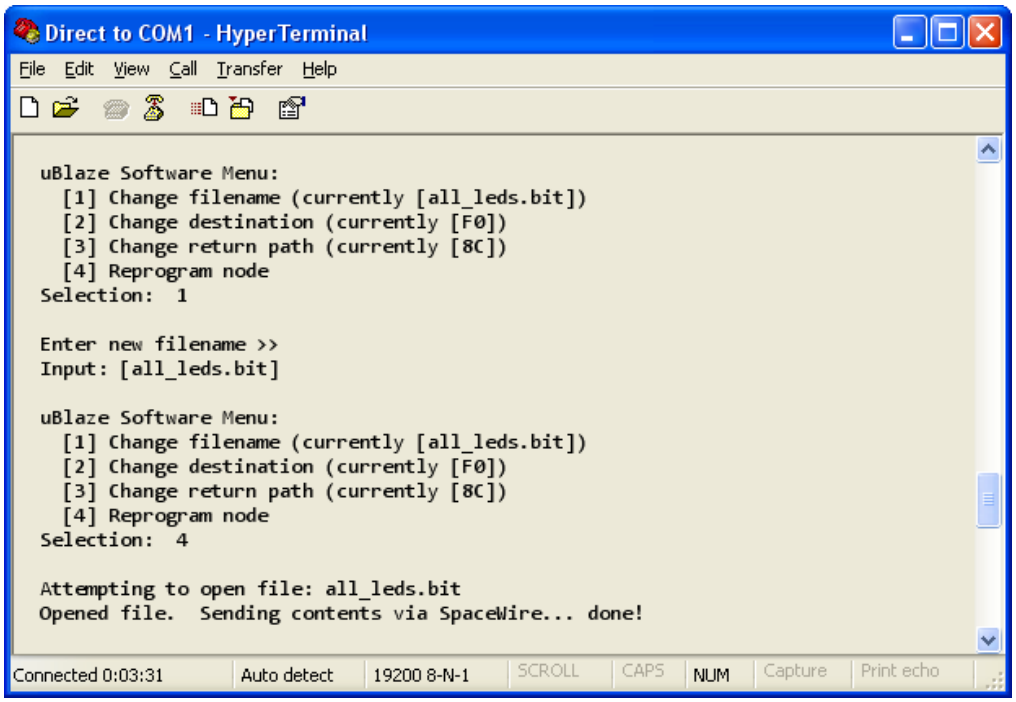

Figure 30. Remote configuration over SpaceWire control interface.

This proof-of-concept successfully configured a remote FPGA using various bitfiles transmitted across a variety of network topologies.

5. HARDWARE DEMONSTRATION CONCLUSIONS

This effort has demonstrated the transport of application layer packets across both RapidIO and SpaceWire networks to a common downlink destination using small topologies comprised of COTS and custom devices.

A complete demonstration that includes one of the researched topologies [18] for this LDRD [1] would require at least 18 to 27 nodes. This demonstration, however, was designed in order to prove the operation of the functions discussed in this report. A larger network would allow for a much more in-depth level of verification, which could make the designs useful in practical application.

The RapidFET and NEX-SRIO debug and verification tools were instrumental in the successful implementation of the RapidIO hardware demonstration. It is highly recommended that any future designers have this test equipment available before creating any custom components that integrate the RapidIO protocol. These tools also proved the benefit of working with standard serial interconnects in that commercial test equipment can be applied to the development effort and not having to rely on custom hardware to provide this capability.

The SpaceWire implementation also successfully demonstrated the transfer and routing of application data packets between multiple nodes. In this exercise, SpaceWire exhibited a number of positive characteristics, including ease of implementation, simple protocol standard, and availability of features desirable for space networks (such as bandwidth throttling to save power). These traits, along with SpaceWire's use in both past and current flight systems, make SpaceWire a strong candidate for use in satellite networks.

In addition to proving the feasibility of both the RapidIO and SpaceWire protocols, this hardware demonstration was able reprogram remote nodes using configuration bitfiles transmitted over the network. This is one of the key features proposed in NBAs, and leveraging this work demonstrates a key component of NBAs that will improve future system reliability and enhance the capabilities of these systems.

The favorable results outlined in this document illustrate the potential use of either RapidIO or SpaceWire in real-world NBAs. Which protocol is used in future satellite architectures will ultimately be determined by a number of factors, including bandwidth requirement, protocol feature set, and resource utilization. In either case, IP has been produced to support data transmission using both protocols that can be leveraged in the development of future systems.

6. REFERENCES

1. Jeffrey L. Kalb, *Modeling and Design of High Speed Networks for Satellite Applications,* SAND2008-5810. Albuquerque, NM: Sandia National Laboratories, September 2008.

2. John M. Eldridge and Jeffrey L. Kalb, *Survey of Communication Protocols for Satellite Payloads,* SAND2008-0254. Albuquerque, NM: Sandia National Laboratories, January 2008.

3. ECSS-E-50-12A, *Space Engineering: SpaceWire – Links, Nodes, Routers, and Networks*. European Space Agency, January 2003.

4. David Heine, Jeffrey L. Kalb, David S. Lee, John M. Eldridge and Eric Ollila, *Network Modeling Study,* SAND2008-6001. Albuquerque, NM: Sandia National Laboratories, September 2008.

5. Xilinx, Inc. *ML325 Characterization Board*. May 2006. (http://www.xilinx.com/products/devkits/HW-V2P-ML325.htm)

6. RapidIO Interoperability Laboratory. 2008. (http://www.rio-lab.com/)

7. Silicon Turnkey Express (STx), *Serial RapidIO Development Platform (SRDP)*. 2008. (http://www.silicontkx.com/srdp.htm)

8. RapidIO Trade Association, *RapidIO Specification v1.3*. 2008. (http://www.rapidio.org/specs/current)

9. Tundra Semiconductor Corporation, *Tsi578 Serial RapidIO Switch User Manual*. November 2007.

10. Tcl/Tk, *Tool Command Language & Toolkit*. 2008. (http://tcl.sourceforge.net/)

11. Xilinx, Inc., *ISE Foundation & Embedded Development Kit v9.2i*. 2008. (http://www.xilinx.com)

12. Fabric Embedded Tools Corp., *RapidFET Professional and RapidFET Probe*. 2008. (http://www.fetcorp.com/)

13. Tektronix, Inc. (http://www.tek.com/)

14. Nexus Technology, Inc., *NEX-SRIO Protocol Analyzer*. 2008. (http://www.busboards.com/products/bus/srio/index.html)

15. Daniel E. Gallegos, Benjamin Welch, Jason Jarosz, Jonathan Van Houten, and Mark Learn, *Soft Core Processing Study for Node Based Architectures,* SAND2008-6015. Albuquerque, NM: Sandia National Laboratories, September 2008.

16. David M. Holman and David S. Lee, *A Survey of Routing Techniques in Store-and-Forward and Wormhole Interconnects,* SAND2008-0068. Albuquerque, NM: Sandia National Laboratories, January 2008.

17. Xilinx, Inc., *Virtex-4 FPGA Configuration User Guide*. April 2008. (http://www.xilinx.com/support/documentation/user_guides/ug071.pdf)

18. David S. Lee and Jeffrey L. Kalb, *Network Topology Analysis,* SAND2008-0069. Albuquerque, NM: Sandia National Laboratories, January 2008.

DISTRIBUTION

1 Leonard G. Burczyk
Mail Stop D440
Los Alamos National Laboratory
Los Alamos, NM 87545

1	MS0343	Jim B. Woodard, 2600
1	MS0406	Toby O. Townsend, 5713
1	MS0501	Steve B. Rohde, 5337
1	MS0503	Mythi M. To, 5337
1	MS0530	Dan E. Gallegos, 2623
1	MS0964	Brian C. Brock, 5733
1	MS0971	Ethan L. Blansett, 5733
1	MS0971	Bob M. Huelskamp, 5730
1	MS0971	Jae W. Lee, 5733
1	MS0972	Kurt R. Lanes, 5560
1	MS0980	Jay F. Jakubczak, 5710
1	MS0980	Matt P. Napier, 5571
1	MS0980	Steve M. Gentry, 5703
1	MS0982	Dan E. Carroll, 5732
1	MS0982	J. Doug Clark, 5732
1	MS0982	Dan Kral, 5732
1	MS0986	Dave M. Bullington, 2664
1	MS0986	Jonathon W. Donaldson, 2664
1	MS0986	Dave Heine, 2664
1	MS0986	Jeff L. Kalb, 2664
1	MS0986	Dave S. Lee, 2664
1	MS0986	J. (Heidi) Ruffner, 2664
1	MS0986	John V. Vonderheide, 2660
1	MS1172	Ron J. Franco, 5415
1	MS1202	Ed J. Nava, 5632
1	MS1235	John M. Eldridge, 5632
1	MS1243	Ray H. Byrne, 5535
1	MS0123	D. Chavez, LDRD Office, 1011
1	MS0899	Technical Library, 9536 *(electronic copy)*


Sandia National Laboratories