Report Number: WUCSE-2004-53

2004-09-15

TCP Processor: Design, Implementation, Operation, and Usage

David V. Schuehler

There is a critical need to perform advanced data processing on network traffic. In order to accom-plish this, protocol processing must first be performed to reassemble individual network packets into consistent data streams representing the exact dataset being transferred between end systems. This task is currently performed by protocol stacks running on end systems. Similar protocol processing opera-tions are needed to process the data on the interior of the network. Given millions of network connections operating on multi-gigabit per second network links, this task is extremely difficult. The TCP-Processor addresses this challenge. It is a hardware circuit designed to... Read complete abstract on page 2.

TCP Processor: Design, Implementation, Operation, and Usage

David V. Schuehler

Complete Abstract:

There is a critical need to perform advanced data processing on network traffic. In order to accom-plish this, protocol processing must first be performed to reassemble individual network packets into consistent data streams representing the exact dataset being transferred between end systems. This task is currently performed by protocol stacks running on end systems. Similar protocol processing operations are needed to process the data on the interior of the network. Given millions of network connections operating on multi-gigabit per second network links, this task is extremely difficult. The TCP-Processor addresses this challenge. It is a hardware circuit designed to perform TCP stream reassembly operations for 8 million bidirectional TCP connections at OC-48 (2.5 Gbps) data rates. This document takes an in-depth look at the TCP-Processor technology, related stream processing applications, and other utilities that support the development of the TCP-Processor.

2004-53

TCP Processor: Design, Implementation, Operation, and Usage

Authors: Schuehler, Dave V.

September 15, 2004

Abstract: There is a critical need to perform advanced data processing on network traffic. In order to accomplish this, protocol processing must first be performed to reassemble individual network packets into consistent data streams representing the exact dataset being transferred between end systems. This task is currently performed by protocol stacks running on end systems. Similar protocol processing operations are needed to process the data on the interior of the network. Given millions of network connections operating on multi-gigabit per second network links, this task is extremely difficult.

The TCP-Processor addresses this challenge. It is a hardware circuit designed to perform TCP stream reassembly operations for 8 million bi-directional TCP connections at OC-48 (2.5 Gbps) data rates. This document takes an in-depth look at the TCP-Processor technology, related stream processing applications, and other utilities that support the development of the TCP-Processor.

TCP-Processor: Design, Implementation, Operation and Usage

David V. Schuehler

Department of Computer Science and Engineering
Applied Research Lab
Washington University
1 Brookings Drive, Box 1045
Saint Louis, MO 63130

WUCSE-2004-53
`http://www.arl.wustl.edu/arl/projects/fpx`

September 15, 2004

Abstract

There is a critical need to perform advanced data processing on network traffic. In order to accomplish this, protocol processing must first be performed to reassemble individual network packets into consistent data streams representing the exact dataset being transferred between end systems. This task is currently performed by protocol stacks running on end systems. Similar protocol processing operations are needed to process the data on the interior of the network. Given millions of network connections operating on multi-gigabit per second network links, this task is extremely difficult.

The TCP-Processor addresses this challenge. It is a hardware circuit designed to perform TCP stream reassembly operations for 8 million bi-directional TCP connections at OC-48 (2.5 Gbps) data rates. This document takes an in-depth look at the TCP-Processor technology, related stream processing applications, and other utilities that support the development of the TCP-Processor.

Contents

List of Figures

1 Introduction

The TCP-Processor technology was born out of the desire to process Transmission Control Protocol (TCP) [1]-based data streams within extensible networking devices. Initial efforts to process TCP packets in a hardware circuit resulted in the development of the TCP-Splitter technology [2]. The TCP-Processor takes the TCP-Splitter technology several steps further and represents a more mature and feature rich implementation of a TCP processing circuit.

This technical report provides detailed information about the TCP-Processor (version 2). The source code and build environment can be retrieved from the URL: `http://www.arl.wustl.edu /projects/fpx/fpx_internal/tcp/source/streamextract_source_v2.zip`.

Figure 1 illustrates the directory structure for this distribution. All source, make, and project files required to compile, simulate, synthesize and build the StreamExtract circuit are included in this distribution. The StreamExtract circuit contains the TCP-Processor logic and supports multidevice TCP stream processing applications. The standard project layout format common to other FPX group projects is used for this project. The list below gives a brief summary of the contents of each subdirectory in the distribution:

`backend`- Contains files needed to perform backend simulations.

`iptestbench`- Contains utilities for generating input files used during simulation.

`sim`- Contains files necessary to compile and simulate the StreamExtract circuit.

`syn`- Contains the Synplicity project used when synthesizing the StreamExtract circuit.

`syn/rad-xcve2000e-64MB`- Contains the files required to perform place and route functions. This resultant BIT file will be deposited in this directory.

`vhdl`- Contains all VHDL source for the StreamExtract circuit. The top level is contained in the `rad_streamextract.vhd` file.

`vhdl/CCP_MODULE`- Contains the source files for the control cell processor. NCHARGE communicates with the CCP to allow remote access to SRAM devices.

`vhdl/MEMMOD`- Contains the manufacturer-supplied source files required to simulate the memory devices used on the FPX platform.

`vhdl/SDRAM_CONTROLLER`- Contains Sarang's SDRAM memory controller. This circuit is documented in a technical report [3].

`vhdl/SRAM_CONTROLLER`- Contains the SRAM memory controller. This circuit is documented in a

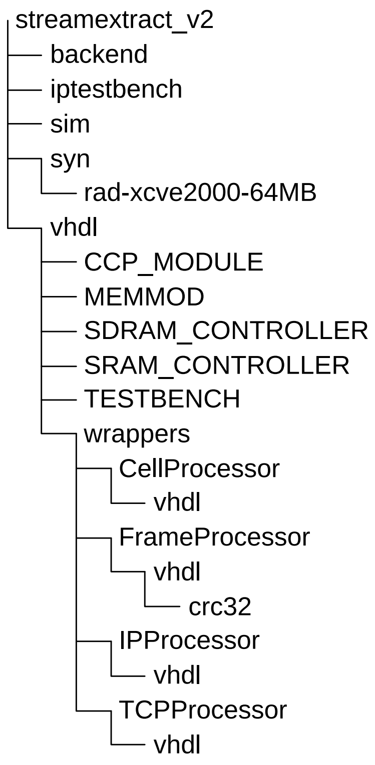


Figure 1: Layout of Directory Tree

technical report [4].

`vhdl/TESTBENCH`- Contains the testbench simulation environment utilized by ModelSim to reconstruct the environment of the FPX platform when performing simulations.

`vhdl/wrappers/CellProcessor/vhdl`- Contains the source for the ATM cell wrapper.

`vhdl/wrappers/FrameProcessor/vhdl`- Contains the source for the AAL5 frame wrapper.

`vhdl/wrappers/IPProcessor/vhdl`- Contains the source for the IP wrapper.

`vhdl/wrappers/TCPProcessor/vhdl`- Contains the source for the TCP-Processor.

From the root directory the **`make compile`** command will compile the source VHDL files, the **`make sim`** command will start up ModelSim to perform functional simulation, the **`make syn`** command will synthesize the VHDL using Synplicity, and the **`make build`** command will perform place and route functions utilizing Xilinx tools.

Section 2 of this paper provides information about the operational environment of the TCP-Processor and the hardware platform utilized for development, debugging and testing of the circuit. Section 3 describes the internal workings of the TCP-Processor in detail. Section 4 describes the StreamExtract circuit, which contains the TCP-Processor logic and supports multi-device TCP stream processing applications. Section 5 describes the TCPLiteWrappers and applications which utilize these wrappers to implement TCP flow processing applications. Section 6 covers the specific steps necessary to compile, simulate, synthesize, place & route, and run the suite of TCP processing circuits. In addition, information about how to integrate solutions with the TCP-Processor is also provided. Section 7 provides information on various methods for generating simulation input files. Section 8 provides concluding remarks. The Appendix covers associated utilities which store, process, chart, and rebroadcast statistics information generated by the TCP processing circuits described in this document.

2 Environment

The Applied Research Laboratory (ARL) at Washington University in St. Louis has a distinguished history of performing high-speed networking research [5]. The TCP-Processor leverages previous research work performed by faculty, staff, and students at this facility[6]. The following subsections briefly describe some of the hardware, software, and circuit designs which were used to support the development of the TCP-Processor.

The TCP-Processor was designed, implemented and tested to operate within the context of the Field-Programmable Port Extender (FPX). While this circuit is not limited to this environment, minor modifications to the TCP-Processor would most likely be required for it to operate on a different hardware platform. For the purposes of this technical report, it is assumed that the FPX platform is the underlying hardware device for the TCP-Processor.

2.1 Washington University Gigabit Switch

The Washington University Gigabit Switch (WUGS) [7] is an eight port ATM switch interconnected by an underlying switching fabric. Data paths on the switch are 32 bits wide. The backplane of the switch was designed to drive each of the eight ports at a data rate of 2.4 Gbps to achieve an aggregate bandwidth of 20

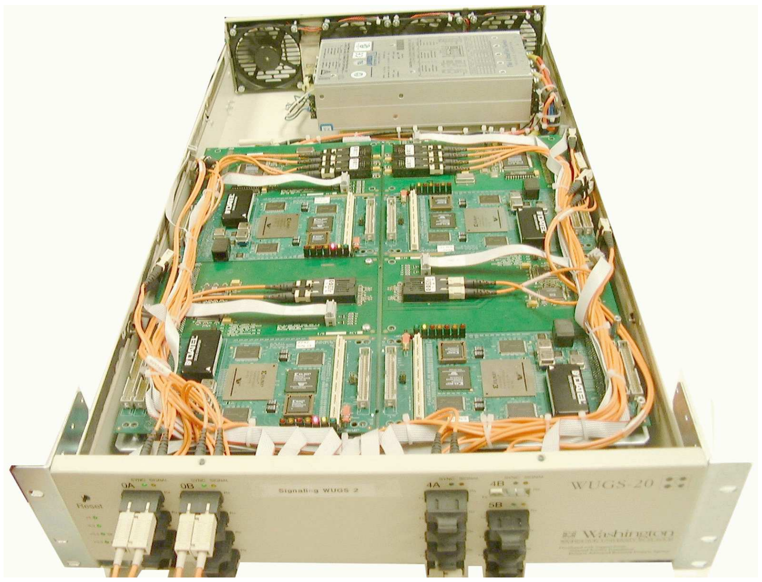


Figure 2: Washington University Gigabit Switch Loaded with Four FPX Cards

Gbps.

Each of the eight ports supports several different types of adapter cards. The current set of adapter cards includes a Gigabit Ethernet line card, a Gigabit ATM line card, an OC3 ATM line card, the Smart Port Card (SPC), the Smart Port Card II (SPC2), and the Field-programmable Port Extender (FPX). The SPC and SPC2 contain a Pentium processor and Pentium III processor respectively, and can execute a version of the Unix operating system which supports software-based packet processing [8]. Figure 2 shows a WUGS switch populated with four FPX cards, two OC-3 line cards, and two GLink line cards.

2.2 Field-programmable Port Extender

The Field-programmable Port Extender (FPX) [9] is a research platform which supports the processing of high-speed network traffic with reconfigurable devices. All processing on the FPX is implemented in reconfigurable logic with FPGAs. One FPGA on the system is called the Reprogrammable Application Device (RAD) and is implemented with a Xilinx Virtex XCV2000E (earlier models of the FPX use the XCV1000E) [10]. The RAD can be dynamically reconfigured to perform user-defined functions on packets traversing through the FPX. The second FPGA device on the FPX is called the Network Interface Device (NID) and is implemented with a Xilinx Virtex XCV600E FPGA. The NID, which is statically programmed at power up, controls data routing on the FPX platform and is also responsible for programming the RAD. The system contains five parallel banks of memory that include both Zero Bus Turnaround (ZBT) pipelined Static Random Access Memory (SRAM) for low-latency access to off-chip memory and Synchronous Dynamic RAM (SDRAM) to enable buffering of a gigabyte of data. Network interfaces allow the FPX to interface at speeds of OC-48 (2.4 gigabits per second). The RAD device can be remotely reprogrammed by sending specially configured control cells to the device. A diagram of the components of the FPX platform can be seen in Figure 3 [11].

2.3 NCHARGE

A suite of tools called NCHARGE (Networked Configurable Hardware Administrator for Reconfiguration and Governing via End-systems) remotely manages control and configuration of the FPX platform [12]. NCHARGE provides a standard Application Programming Interface (API) between software and reprogrammable hardware modules. Using this API, multiple software processes can communicate to one or more FPX cards using standard TCP/IP sockets.

2.4 FPX-in-a-Box

The FPX platform can be used either in conjunction with a WUGS switch or independently as part of a standalone solution. The FPX cards have a network interface port on both the top and bottom of the card, enabling stacking of other FPX or line cards under and over them. The FPX-in-a-box system provides a simple backplane which supports two stacks of FPX cards with a line card placed at the top of each stack. The provides a small footprint system which is capable of performing complex network processing at OC-48 data rates without requiring a underlying switch fabric. Systems needing complex flow processing can be implemented by stacking multiple FPX cards and distributing circuits across the multiple FPGA devices.

2.5 Protocol Wrappers

FPX cards process Internet packets using a set of Layered Protocol Wrappers [16, 13, 14]. Each protocol wrapper performs processing on a different protocol layer for inbound and outbound traffic. These protocol wrappers create a framework that allows upper layer protocol processing to be isolated from any underlying protocols. The protocol wrapper library includes an ATM Cell wrapper, an AAL5 Frame wrapper, an IP

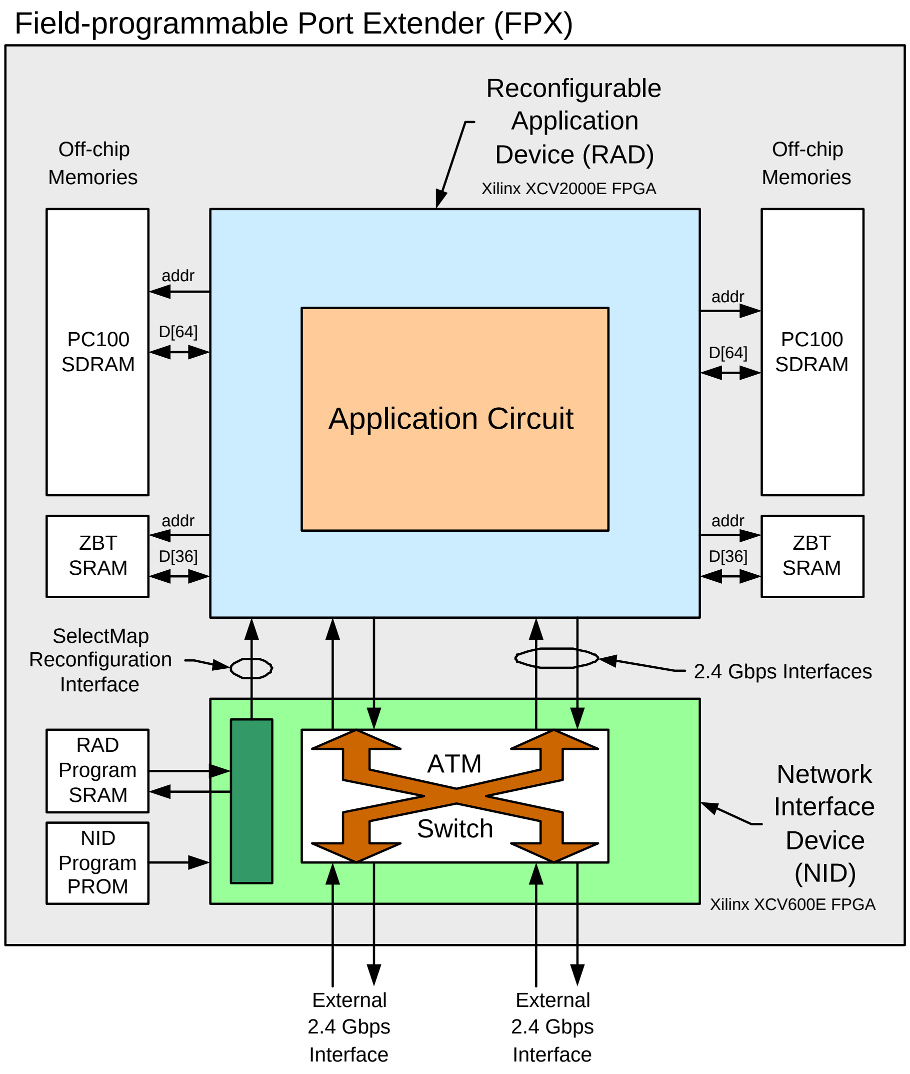


Figure 3: Field Programmable Port Extender

protocol wrapper and a UDP protocol wrapper. The Cell Wrapper validates the Header Error Checksum (HEC) field in the ATM cell header and performs routing based on the cell's Virtual Channel Identifier (VCI) and Virtual Path Identifier (VPI). The Frame Wrapper performs Segmentation and Reassembly (SAR) operations converting cells into packets and packets back into cells. The IP Wrapper performs processing of IP packets. The TCP-Processor circuit interfaces with the IP, Frame and Cell Wrappers for processing network traffic.

The IPWrapper sends and receives IP packets utilizing a 32-bit wide data bus for carrying data and several associated control signals. These control signals include a start of frame (SOF) signal, a start of IP header (SOIP) signal, a start of payload (SOP) signal, and an end of frame (EOF) signal. Packets themselves can be broken down into five sections as described in Table 1.

Data	Control signals	Comment
ATM VCI	SOF	first word of packet
preamble	dataen	zero or more words inserted before the start of the packet
IP header	dataen, SOIP, EOF	IP header information
IP payload	dataen, SOP, EOF	IP payload information
trailer	dataen	contains AAL5 trailer fields

Table 1: IP packet contents

The IPWrapper is hard-coded to process only the traffic that enters on VCI 50 (0x32) or VCI 51 (0x33). Because of this implementation feature, all network traffic processed by the IPWrapper must arrive on VCI 50 (0x32) or VCI 51 (0x33). This includes TCP traffic to be processed by the TCP-Processor and control traffic used to remote command and control various circuit designs. In order to process data on a VCI other than 50 (0x32) or 51 (0x33), VHDL code changes will be required in the lower layered protocol wrappers.

3 Internals

The TCP-Processor is a subcomponent of the StreamExtract application which will be discussed in the next section. The TCP-Processor interfaces to the previously mentioned layered protocol wrappers and provides protocol processing for TCP packets. More specifically, the TCP-Processor is concerned with reassembling application data streams from individual TCP packets observed on the interior of the network. This section will focus on the specifics of the TCP-Processor.

The source code for TCP-Processor is located in the directory `vhdl/wrappers/TCPProcessor /vhdl` found in the `streamextract_source_v2.zip` distribution file. The following VHDL source modules can be found there and comprise the core of the TCP-Processor technology:

`tcpprocessor.vhd`- Top level TCP-Processor circuit. The external interface includes configuration parameters, clock and control signals, inbound and outbound 32-bit UTOPIA interfaces which provides access to raw ATM cells, two separate SRAM interfaces, a single SDRAM interface, and four output signals tied to LEDs. The TCPProcessor component interfaces the lower layer protocol wrappers with the core TCP-Processor.

`tcpproc.vhd`- This module encapsulates the individual components which make up the TCP-Processor. Network traffic is routed between the external interfaces and the various subcomponents.

`tcpinbuf.vhd`- The TCPInbuf component provides inbound packet buffering services for the TCP-Processor when back pressure or flow control is driven by downstream components. This module also ensures that data loss occurs on packet boundaries.

`tcpengine.vhd`- The TCPEngine component performs TCP processing. It communicates with the state store manager to retrieve and store per-flow context information.

`statestoremgr.vhd`- The StateStoreMgr context storage and retrieval services. A simple external interface is exposed and all complex interactions with SDRAM are handled here.

`tcprouting.vhd`- The TCPRouting module routes packets previously processed by the TCPEngine to either the client monitoring circuit, the outbound TCPEgress module, or to the bit bucket. Additional control signals indicate how each packet should be routed.

`tcpstats.vhd`- The TCPStats component collects statistical information from other components and maintains internal event counters. On a periodic basis, a UDP statistics packet is generated and sent to the configured destination address.

`tcpegress.vhd`- The TCPEgress module merges statistics traffic from the TCPStats component, bypass traffic from the TCPRouting component, and return traffic from the client monitoring application. TCP checksum values can also be regenerated for TCP packets returning from the client monitoring application to support flow modification.

The components of the TCP-Processor directly correspond to the various source files. To simplify the layout of the circuit, each component is completely contained in a single, unique source file. Figure 4 diagrams the components of the TCP-Processor and their relative significance in the VHDL hierarchy.

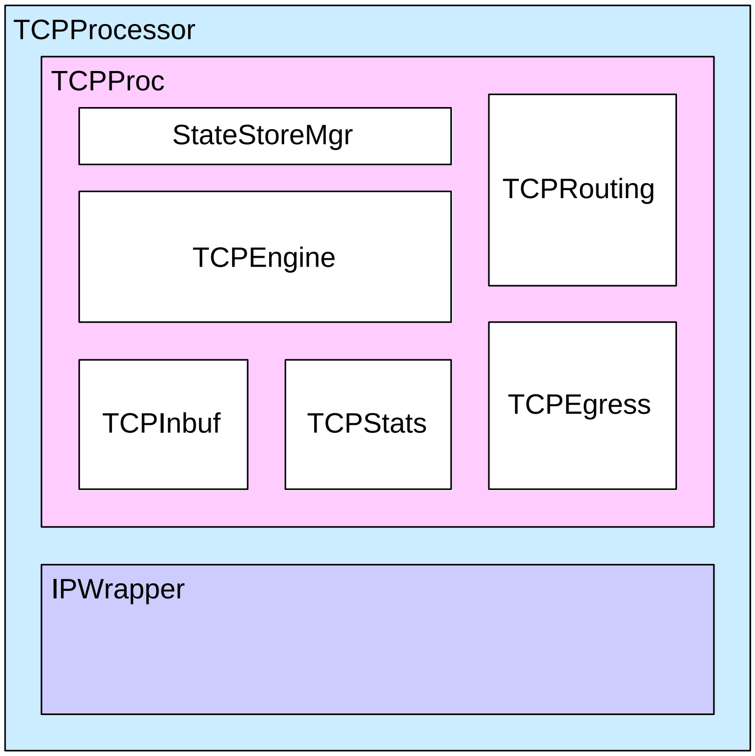


Figure 4: Hierarchy of TCP-Processor Components

3.1 Endianness

The TCP-Processor was developed with little endian constructs. In all signal vectors, the least significant bit is the lowest bit of the vector (usually bit zero). This is the most intuitive method for representing data and is used throughout the TCP-Processor and associated circuits. The TCP-Processor has been constructed to work with network headers using little endian constructs, even though network byte order is a big endian format.

3.2 Packet Parameters

The TCP-Processor supports packet lengths of 1500 bytes or less. When the circuit is presented with packets larger than 1500 bytes, the viability of the operation of the circuit is not guaranteed. The circuit could likely handle larger packets, but would be subject to lockups when packet lengths exceed the buffering capacity of internal FIFOs.

The TCP-Processor does not support IP fragments. IP defragmentation is a lower layer protocol function which should occur within the IPWrapper. The IPWrapper does not contain any logic to reassemble IP fragments into complete packets, or even validate the IP packet length. The behavior of the circuit is currently undefined when IP fragments are processed.

3.3 Flow Control

Within the confines of the TCP-Processor, it is assumed that whenever the TCA signal is driven low, the sending component can continue to send data until the end of the current packet is reached. This behavior contrasts with how the other protocol wrapper circuits manage flow control signals. The advantage of this behavior is that it simplifies the logic associated with processing TCA signals. The disadvantage is that it requires larger buffers and/or FIFOs in order to store the additional data after deasserting TCA. This behavior is internal to the TCP-Processor and affects components within the TCP-Processor and any client monitoring application which interfaces with the TCP-Processor. When interfacing with the IP Wrapper, the TCP-Processor conforms to its flow control semantics.

3.4 External Memories

The TCP-Processor utilizes SDRAM memory module 1 to maintain per-flow context information. This task completely consumes this memory module and it should therefore not be used for any other purpose. SDRAM memory module 2 is not used by the TCP-Processor and is available for other features, such as IP packet defragmentation or TCP packet reordering. The two SRAM banks are used to capture packet data for debugging purposes. By eliminating the debugging feature, both of these high speed memory devices could be used for other purposes.

3.5 Configuration Parameters

The TCP-Processor uses several configuration parameters to control its operational behavior. The following list describes each of those parameters in detail, along with possible values and the effect that parameter has on the operation of the TCP-Processor circuit:

num_pre_hdr_wrds- Passed to the IPWrapper component where it is used to determine the start of the IP packet header. Known possible values for this parameter are 0 - for ATM environments, 2 - for ATM environments with a two word LLC header, 4 - for gigabit Ethernet environments, and 5 - for gigabit Ethernet environments with a VLAN tag.

simulation- Indicates whether or not the TCP-Processor circuit should operate in simulation mode. When set to 1, memory initialization (the zeroing of external memory) is skipped. Additionally, all accesses to external SDRAM are performed at address location zero. This parameter enables the quick simulation of network traffic because the initialization of large external memories is avoided. This parameter should be set to 0 when building circuits to operate in hardware.

skip_sequence_gaps- When set to 1, this parameter indicates that the TCPEngine should skip sequence gaps and track the highest sequence number associated with every flow. If a sequence gap occurs, the TCPEngine sets the NEW_FLOW flag to indicate that data in this packet represents a new stream of bytes. The FLOW IDENTIFIER and the FLOW INSTANCE NUMBER will remain the same as the previous packet associated with the same flow. This information can be used by the monitoring application to determine the occurrence of a sequence gap. When set to 0, the TCP-Processor drops out-of-sequence packets until packets with the appropriate sequence number arrive.

app_bypass_enabled- Utilized by the TCPRouting module which determines where to route packets. When set to 1, retransmitted packets, non-classified TCP packets (see next configuration parameter), and non-TCP packets are allowed to bypass the client monitoring application and pass directly to the TCPEgress component. When this parameter is set to 0, all packets are routed to the client monitoring application.

classify_empty_pkts- Indicates whether or not TCP packets which contain no user data should be classified. Performance of the circuit can be improved by avoiding classification of these packets because it takes longer to classify a packet and associated it with a flow than it takes for these small packets to be transmitted on an OC-48 (2.5Gbps) link. When enabled, all TCP packets are classified. This parameter should be enabled if the client monitoring application is tracking the connection setup/teardown sequences or is trying to process bidirectional traffic and match acknowledgment numbers with corresponding sequence numbers of traffic in the opposite direction.

cksum_update_ena- When set to 1, the TCPEgress component will validate the TCP checksum for all TCP packets passed in by the client monitoring application. If the TCP checksum value is determined to be incorrect, a correcting suffix is added to the end of the packet which will replace the erroneous checksum value with the proper checksum value. When set to 0, no checksum processing is performed by the TCPEgress component.

stats_enabled- Indicates whether or not a statistics packet should be generated at a periodic time interval. When set to 1, the generation of statistics packets is enabled and conforms to the following configuration parameters. When set to 0, statistics information will not be kept nor transmitted.

stats_id- This 8-bit wide configuration parameter can be used to differentiate statistics generated by multiple TCP-Processors. For example, a certain configuration consists of three different instances of the TCP-Processor circuit. All three are sending statistics information to the same collection facility. This configuration parameter differentiates the statistics information among the three different TCP-Processor circuits.

stats_cycle_count- This 32-bit wide configuration parameter indicates the interval, in clock cycles, between the sending of a UDP packet containing statistics information collected during the previous collection interval. This parameter is highly dependant on the frequency of the oscillator used to drive the RAD. During simulations, this parameter can be set to a very small number so that statistics packets are generated on a much more frequent basis.

stats_vci- This 8-bit wide configuration parameter specifies the VCI on which the statistics packet should be generated. By altering this value, statistics packets can be routed differently than normal monitored traffic.

stats_dest_addr- This 32-bit wide configuration parameter specifies the destination IP address for the statistics packets.

stats_dest_port- This 16-bit wide configuration parameter specifies the destination UDP port number for the statistics packets.

3.6 TCP Processor

The TCPProcessor component provides the top level interface for the TCP-Processor and is implemented in the `tcpprocessor.vhd` source file. Within this component, the lower layered protocol wrappers are integrated with the various TCP processing components. Figure 5 contains a layout of the TCPProcessor circuit along with groupings of the main signal bundles and how they are connected between external interfaces and internal components.

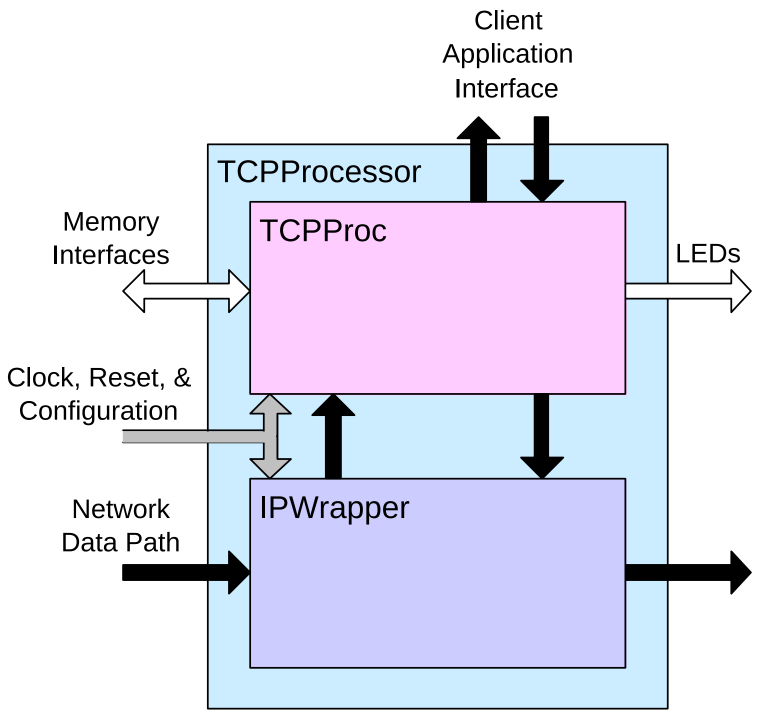


Figure 5: TCPProcessor Layout

3.7 TCP Proc

The TCPProc component provides an integration point for the core components of the TCP-Processor circuit and is implemented in the `tcpproc.vhd` source file. The TCP-Processor has been broken into several manageable modules in order to isolate the various functionalities of the components. It consists of the following six components: TCPInbuf, TCPEngine, StateStoreMgr, TCPRouting, TCPEgress, and TCPStats. Well-defined interfaces allow data to transition between these components. The interface specification will be described in later sections. Figure 6 describes a layout of the TCPProc circuit along with some of the main interconnections between components.

Two SRAM memory interfaces are connected through the external interface of the TCPProc component. Inside the TCPProc component, there are three possible connection points for the two SRAM devices: two in the TCPInbuf module and one in the TCPEgress module. In order to accommodate this difference, the

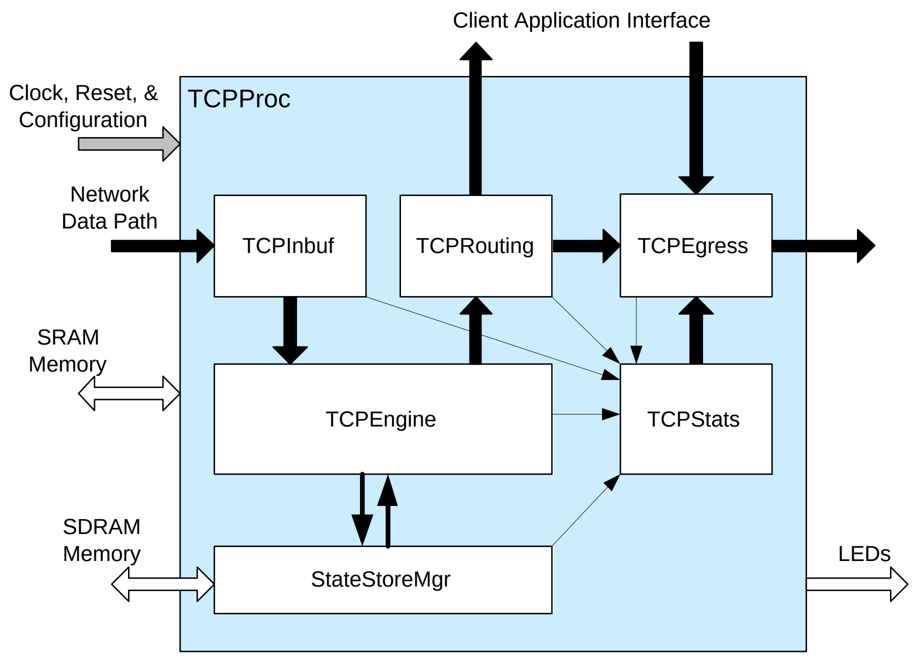


Figure 6: TCPProc Layout

TCPProc maintains a virtual NOMEM1 memory device. Selected NOMEM1 signals are driven to give the appearance that the memory device is always available, but writes to the device never take place and reads from the device always result in the value of zero. By changing the connections within the TCPProc circuit, selected subcomponent interfaces can be connected to the external memory devices and others can be connected to the virtual device. This allows network traffic to be captured to memory at various locations within the circuit to aid in debugging.

3.8 TCP Input Buffer

Network traffic that enters the TCP-Processor is first processed by the TCPInbuf component and is implemented in the `tcpinbuf.vhd` source file. It provides packet buffering services to the TCP-Processor when there are periods of downstream delay and back pressure is applied by downstream components via a flow control signal. In addition, the TCPInbuf component ensures that only fully formed IP packets are passed into the downstream TCP-Processor components. The TCPInbuf component always asserts the up-

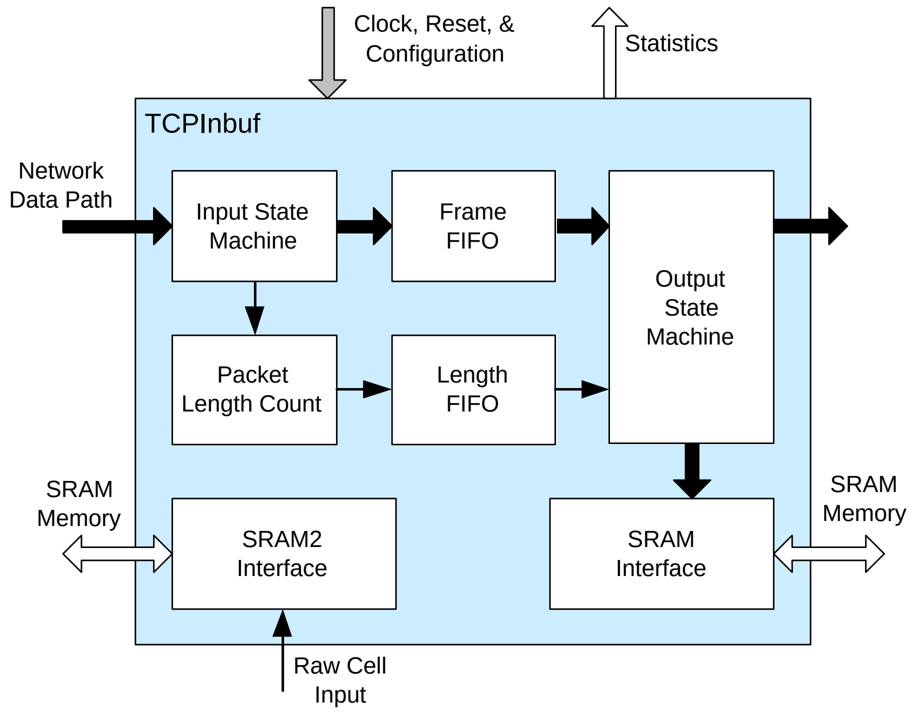


Figure 7: TCPInbuf Layout

stream flow control signal, indicating that it is always ready to receive data. If its internal buffers are filled to capacity, then TCPInbuf manages the dropping of packets form the network instead of deasserting the TCA flow control signal. Figure 7 describes the layout of the TCPInbuf component.

As network traffic enters the TCPInbuf component, it is first processed by the input state machine. The state machine determines whether the packet should be dropped (because the Frame FIFO is full) or should be inserted into the Frame FIFO. A separate packet length counting circuit counts the number of 32-bit words associated with the packet which are inserted into the Frame FIFO. At the end of the packet, this length value is written to the Length FIFO. If only a portion of the packet was successfully written to the Frame FIFO (because the FIFO filled up while inserting words), then only the length associated with the number of words actually inserted into the Frame FIFO is passed to the Length FIFO.

The output state machine can utilize two different methods to retrieve data from the FIFOs and forward it downstream: uncounted mode and counted mode. The uncounted mode is used when there are no packets queued in the Frame FIFO. This mode has a performance benefit because it does not induce a store and forward delay for the packet while the packet length is being computed. When the output state machine detects a non-empty Frame FIFO and empty Length FIFO, it will enter the uncounted processing mode. Packet data words are clocked out of the Frame FIFO and passed to the output interface of the TCPInbuf component. When the final word of the packet is read from the Frame FIFO, then the length of the packet is read from the Length FIFO.

After there has been some downstream delay in the system and one or more packets are queued up in the Frame and Length FIFOs, the counted mode is entered. This mode is also entered when the output state machine is in the idle state and data is present in both the Length and Frame FIFOs. The packet length is first read from the Length FIFO. Bits 8 through 0 indicate the length in units of 32-bit words. Bit 9 is a special drop flag which indicates whether or not the packet should be dropped. This flag is set whenever an incomplete packet is inserted into the Frame FIFO. If the drop flag is set, then the specified number of words is read from the Frame FIFO and discarded. If the drop flag is zero, the the specified number of words is read from the Frame FIFO and passed to the output interface of the TCPInbuf component.

The packet storage and SRAM signals manage two separate SRAM devices used to capture packet data for debugging purposes. After reset, both SRAM memory devices are initialized to zero. The control circuits are configured to use locations 1 through 262,143 as a ring buffer. Packets passed to the outbound interface are written to the first SRAM interface and inbound ATM cells from the external RAD interface are written to the second SRAM interface. When the termination criteria has been met (check source for logic), then the memory update logic is put into a mode where it stores the current memory write address to location zero. In this manner, the 262,143 32-bit packet data words can be captured prior to the detected event. Packets (frames) and cells are stored in the standard format required by the SRAMDUMP utility. This layout of the 36-bit SRAM memory for both of these formats can be seen in Figure 8.

3.9 TCP Engine

The TCPEngine component is the workhorse component of the TCP-Processor and is implemented in the `tcpengine.vhd` source file. All of the complex protocol processing logic resides here. Figure 9 shows a breakdown of the major logic blocks and data flow through the component. Network traffic travels left to right. Flow classification and interactions with the state store manager are handled at the bottom of the diagram. The input state machine and output state machine are noted by separate boxes at the left and right

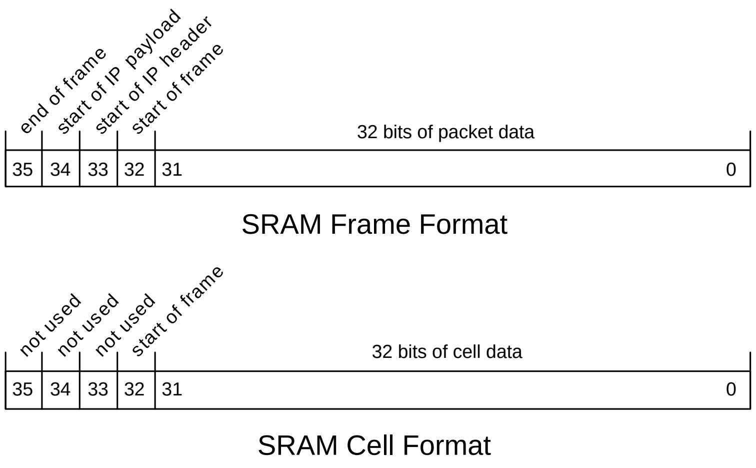


Figure 8: SRAM data Formats

sides of the figure. Additional state machines manage interactions with the state store manager. These are contained in the state store request/response/update block of logic.

This component has evolved over time and contains remnants of logic for features which are no longer utilized. The direction signal and the multiple hash computations are just a couple examples of this extra logic. This VHDL code remains in case it is required again in the future. Since this document describes a version of the TCP-Processor which contains this unused logic, its presence is noted.

The first series of processes described in the source file generate event signals which are passed to the TCPStats component. These signals pulse for one clock cycle whenever the associated event counter should be incremented.

When IP packets enter the TCPEngine, the input state machine processes them first. It tracks the main processing state for the TCPEngine component while processing inbound packet data. The individual states refer to the part of the packet being processed. IP header processing states include version, identifier, protocol, source IP, destination IP, and option fields. TCP header processing states include ports, sequence number, acknowledgment number, data offset, checksum, and option fields. Data0 and data1 states represent processing of TCP payload data. The copy state is entered when a non-TCP packet is encountered and the circuit switches to a mode where it just copies the packet to the Frame FIFO and no other processing is performed. The length and CRC states refer to the AAL5 trailer fields which are tacked on to the end of the

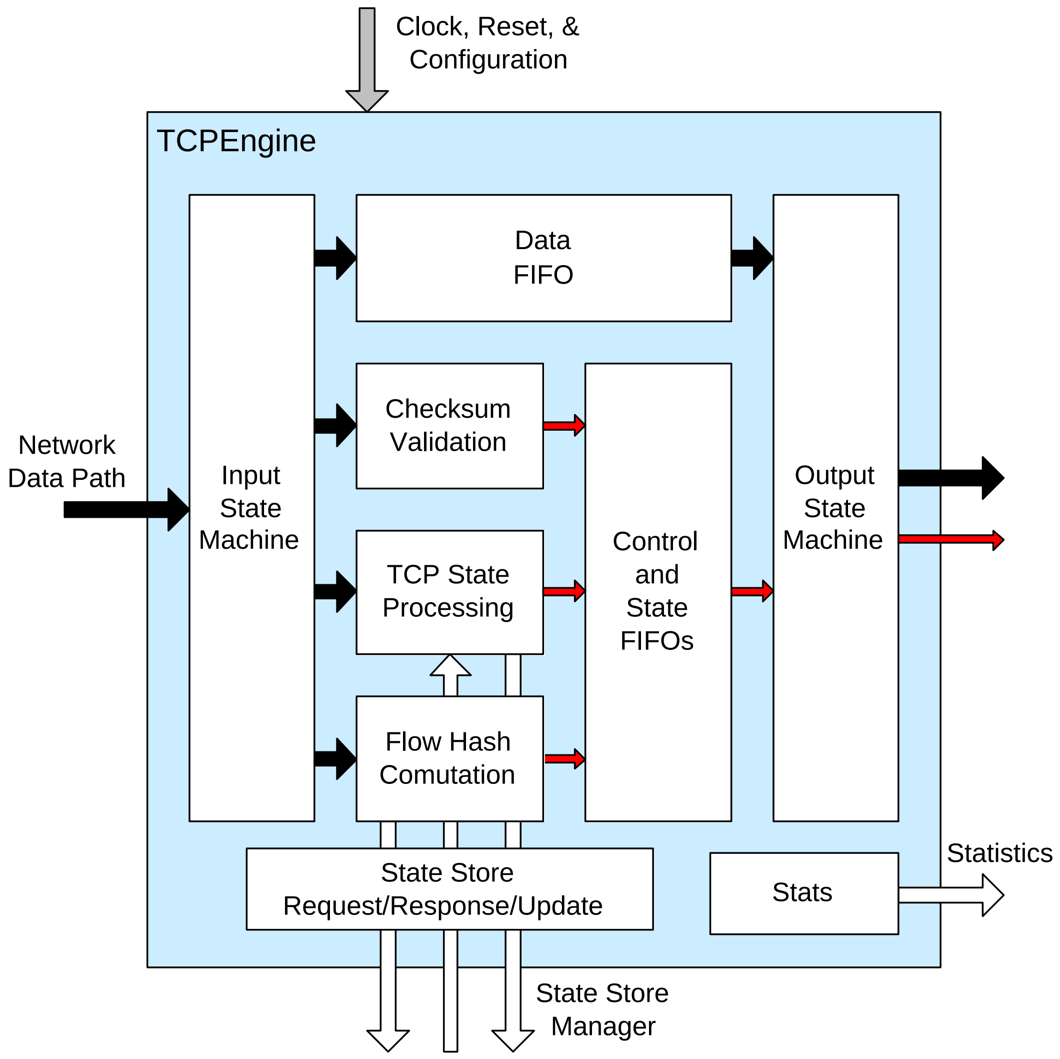


Figure 9: TCPEngine Layout

packet. The WFR state is entered when waiting for a response from the state store manager.

The next section of logic deals with extracting protocol-specific information from the protocol headers. This information includes the IP header length, an indication of whether or not the protocol is TCP, the TCP header length, the current header length, the value of various TCP flags, the source IP address, the destination IP address, and the TCP port numbers.

Next a hash value is computed based on the source IP address, the destination IP address, and the source and destination TCP ports. Many different hashing algorithms have been used throughout the development of the TCP-Processor and source code still exists for several of them. The current hash scheme is hash3. This involves combining the results of a CRC calculation over the various input values. The hash algorithm is constructed such that packets in the forward and reverse directions hash to the same value. This is accomplished by XORing the source fields with the destination fields in an identical manner. The resultant hash value is 23 bits wide, which corresponds to 2^{23} or 8,388,608 unique hash values. This hash value is passed to the state store manager which performs the flow classification operation.

The next section of logic deals with operations required to initiate a flow lookup operation via the state store manager. A request state machine handles the required state transitions. After initiating the request, the state machine enters a hibernate state waiting for the end of input processing for the current packet. This ensures that multiple state store manager lookup requests are not made for one single packet. The sequence of data words which are passed to the state store manager when initiating a request for per-flow context information is listed in Table 2.

Request word	Request data (32 bit field)
1	”000000000” & computed hash value
2	source IP address
3	destination IP address
4	source TCP port & destination TCP port

Table 2: State Store Request Sequence

The following section contains logic to manage the per-flow context data returning from the state store manager. A response state machine tracks this response data. There are three paths that can be traversed through the state machine. The first is entered when a new flow context record is returned from the state store manager, the second corresponds to a valid flow context lookup of an existing flow, and the third is entered when a per-flow context lookup is not performed and a place holder is required. Table 3 shows the state transitions along with the data returned from the state store manager. Bit 31 of the first response word indicates whether or not the response corresponds to a new flow (value 0) or an existing flow (value 1). The returned flow identifier is a 26-bit value. The low two bits of the flow identifier (bits 1-0) are always zero. Utilizing the whole flow identifier as a memory address for the SDRAM on the FPX platform provides access to a separate 32-byte section of memory for each unique flow. Bit 2 of the flow identifier is a direction bit. If only traffic propagating the network in a single direction is being monitored, then the value of this bit will always be zero. This also implies that 64 bytes of data can be stored for each unidirectional flow. If bidirectional traffic is being monitored, then the flow identifiers of outbound and inbound traffic associated with the same TCP connection will differ only by the value of this bit. There is no correlation between the value of this bit and the direction of the traffic. The only conclusion that can be drawn is that packets with flow identifiers that have the bit set are traversing the network in the opposite direction of packets with flow identifiers that have the bit cleared.

	New flow		Existing flow	
Response word	State	Response data (32 bit field)	State	Response data (32 bit field)
1	Idle	”000000” & flow identifier	Idle	”100000” & flow identifier
2	Instance	x”000000” & instance	Sequence	current sequence number
3			Instance	x”000000” & instance

Table 3: State Store Response Sequence

The next section of logic contains processes which perform the protocol-specific processing functions of the TCP-Processor. This includes a process which loads state/context information into the app state FIFO. When the NIL states are entered, zero values are written into this FIFO instead of valid data. This occurs when processing non-TCP packets. The contents of this FIFO will eventually be passed to the client monitoring application via the flowstate signal lines. Four data words are written to this FIFO for every packet processed. The data words either contain zeros or valid context information, depending on the type of packet being processed and the TCP-Processor configuration settings. Other logic in this section of code computes a byte offset to the first byte of new data for the flow and computation of the next sequence expected sequence number.

Logic also exists to send updated per-flow context information to the state store manager. The updated context information includes the next expected sequence number for the flow and an indication of whether or not the connection is being terminated. A signal can also be passed to the state store manager indicating that an update is not being generated and the state store manager should return to its request processing state.

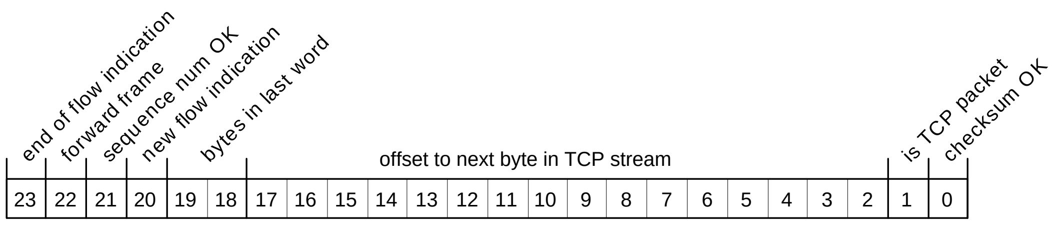


Figure 10: Control FIFO data format

Signals also exist which indicate whether or not this packet is associated with a new flow, keep track of the flow identifier, and compute the length of the TCP payload. The checksum calculation logic comes next in the source file. This logic selects the appropriate words of the IP header, TCP header, and TCP payload to be used in the TCP checksum calculation.

The following section of logic writes control information into the control FIFO. The control FIFO contains information relating to the validity of the TCP checksum, an indication of whether or not this is a TCP packet, the offset to the next byte in the TCP data stream, the number of bytes in the last data word of the packet, a new flow indication, an indication of whether or not the sequence number is in the proper range, an indication of whether or not the packet should just be forwarded (i.e. for retransmitted packets), and an end-of-flow indication. The layout of this data can be seen in Figure 10.

The next process is associated with writing packet data into the data FIFO. The data inserted into this FIFO includes 32 bits of packet data, a start of frame indication, a start of IP header indication, an end of frame indication, a start of IP payload (start of TCP header) indication, a TCP stream data enable signal, a regular data enable signal, and a valid bytes vector indicating how many bytes contain stream data. The valid bytes vector is offset by one, so the value ”00” indicates one valid byte, ”01” indicates two valid bytes, ”10” indicates three valid bytes, and ”11” indicates four valid bytes.

The remainder of the TCPEngine VHDL code is responsible for retrieving data out of the three FIFOs, performing some final data modifications, and passing the information out of the TCPEngine component and onto the TCPRouting component. This whole process is driven by the output state machine which, upon detecting a non-empty control FIFO and app state FIFO, initiates the sequence of events to read information from the FIFOs and pass it to the external interface. Control signals are sent to the control, data, and app state FIFOs to start extracting their contents. As data is retrieved from the data FIFO, computations are

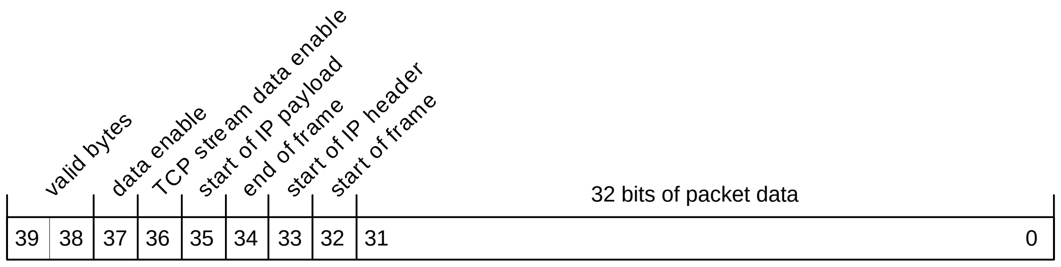


Figure 11: Data FIFO data format

carried out to generate the valid_bytes signal which indicates the number of valid TCP bytes (zero, one, two, three, or four) contained on the packet data bus. Data read from the app state FIFO is passed to the flowstate vector. An offset mask and a length mask are added to this set of data which specify a mask of valid bytes for the first and last words of the stream data that should be considered part of the TCP data stream by the client monitoring application. The offset signals are used to determine where in the packet the client monitoring application should start processing TCP stream data.

3.10 State Store Manager

The StateStoreMgr component provides simple interfaces for accessing and updating per-flow context information stored in external SDRAM and is implemented in the `statestoremgr.vhd` source file. The StateStoreMgr provides two separate interfaces for accessing per-flow context information. The first interface is used by the TCPEngine. A per-flow hash value is presented along with the source IP, destination IP, and source and destination TCP ports. The StateStoreMgr is responsible for navigating to the existing context record for that flow, or allocating new memory to hold context information for the flow. The second interface is unused. It was originally developed to provide the back end of the TCP-Processor access to per-flow context information and supported flow blocking and application state storage services. This second interface is not utilized in this version of the TCP-Processor, but much of the logic remains in place for possible future use.

A version of the TCP-Processor which supports flow blocking and application state storage can be found at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp /source/streamcapture_source.zip`. Please note that the StreamCapture version of the TCP-

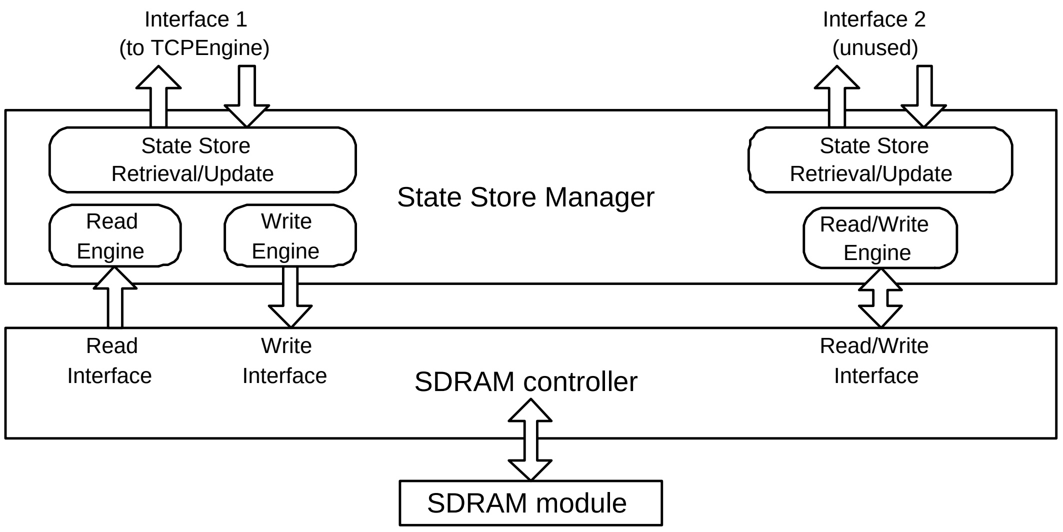


Figure 12: Layout of StateStoreMgr

Processor is an older version of the source and is not documented here.

By separating the logic which manages the per-flow context memory, different lookup, retrieval and memory management algorithms can easily be implemented without requiring complex changes to the TCPEngine. The SDRAM memory controller utilized by the StateStoreMgr (developed by Sarang Dharmapurikar) provides three separate interfaces: a Read only interface, a Write only interface and a Read/Write interface. Figure 12 shows the layout of the StateStoreMgr state machines and their interaction with other components. As previously stated, the second request/update interface is unused. The data bus to the SDRAM controller is 64 bits wide and the data bus exposed by the StateStoreMgr is 32 bits wide. In order to handle the transition between the two different bus widths, an separate 8x64 dual-ported RAM block is utilized in each of the interfaces.

The first process of the `statestoremgr.vhd` source file generates pulses relating to internal events which are passed to that TCPStats module which maintains statistical counters. The statistical events include the occurrence of a new connection, the occurrence of a reused connection (i.e. an active flow context record has been reclaimed and is being used to keep state information for a different flow), the occurrence of a terminated flow (i.e. an existing TCP connection has been closed and no longer requires state storage resources).

The remainder of the StateStoreManager logic is divided into two major sections. The first deals with frontside processing associated with Interface 1 and the second deals with backside processing associated with Interface 2. Interface 1 is connected to the TCPEngine and performs the flow classification, context storage and retrieval operations required by the TCP-Processor. Interface 2 is not used in this instance, but contains logic for performing flow blocking and application state storage services. Enhancements have been made to Interface 1 which have not been incorporated into Interface 2. Before attempting to utilize Interface 2, the inconsistencies between the two interfaces will have to be resolved.

The operations of Interface 1 are driven by the request1 state machine. Upon detecting a request, the state machine traverses through states to save the additional request data. It enters a delay state waiting for a memory read operation to complete. Once data is returned from external memory, a sequence of states are traversed which are responsible for returning results to the requestor. There is also a state to handle the processing of a context update message. There are extra states contained within this state machine which are never reached. These states support the retrieval of client monitoring application context information which is not supported in this version of the TCP-Processor. The next four processes capture flags which are stored in SDRAM. Two of these flags indicate whether the inbound traffic is valid and whether the outbound traffic is valid. The other two flags indicate whether or not the inbound or outbound flow has been terminated. These four signals help the StateStoreMgr to determine the operational state of the context record of a particular TCP flow (i.e. whether or not the record is idle, active for inbound traffic, active for outbound traffic, or whether traffic in either direction has been shut down). It is important to note that the inbound and outbound traffic directions do not correspond to the specific direction of the traffic, but just that inbound traffic is moving in the opposite direction of outbound traffic. Across multiple TCP flow context records, the inbound traffic in each of these flows may actually be moving through the network in different directions.

The subsequent two processes handle performing the exact flow context matches between the context record stored in external memory and the four-tupple of source IP address, destination IP address, and source and destination TCP ports passed in by the requestor. The match1 circuit is performing the match for outbound traffic and the match2 circuit is performing the match for inbound traffic (i.e. the match2 circuit swaps the source and destination parameters).

The following processes store individual data items from the context record returned from external memory. The current sequence numbers for both outbound and inbound traffic are stored along with the instance identifier for that flow. As the per-flow context data is clocked in from memory, these data values are pulled off of the SDRAM read data bus. The instance identifers are maintained for each flow record. Every time a different TCP connection is associated with a flow record, the instance identifier of that record is incremented. This provides a simple mechanism for downstream stream monitoring clients to quickly identify when a flow record is reclaimed and a different TCP connection is mapped to the same flow identifier (ie, memory record).

The next set of processes stores data items passed in as part of the initial request or computed along the way. The flow hash value, one of these data items, is used as the initial index into the state store manager hash table. The direction flop determines the direction of the flow based on which of the two flow matching processes returns a successful match. The upd_direction stores the packet direction during update processing. This allows the request engine to start processing another lookup request. As the source IP addresses, destination IP addresses, and the TCP port values are passed in from the requestor, they are saved and used in later comparisons by the match processes.

The response1 engine is responsible for generating a response to the requesting circuit. It generates response data based on the current processing state, results of the match operations, and the value of the valid flags. The response sequence differs depending on whether or not there was a successful per-flow record match. If an existing flow record is not found, a flow identifier and an instance number are returned. If an existing flow record is found, the current sequence number is also returned. Table 3 shows the returned data in both of these scenarios.

A read state machine manages the interactions with the SDRAM controller read interface. It sequences through PullWait, TakeWait and Take states as defined by the memory controller (developed by Sarang Dharmapurikar). Prepare and Check states ensure that the frontside and backside engines are not trying to operate on the same per-flow record at the same time. Since the backside engine is not used in this incarnation of the TCP-Processor, these states will never be entered. The read engine uses the aforementioned states and drives the SDRAM controller signals to affect a read operation.

The memory layout of a bidirectional per-flow context record is shown in Figure 13. There are many unused bytes in the structure which can be utilized in conjunction with future circuit enhancements. The source IP address, the destination IP address and the source and destination TCP ports are stored within the per-flow record so that (1) an exact match can be performed to ensure that the per-flow record corresponds to the flow being retrieved and (2) the direction of the traffic (either outbound or inbound) can be determined.

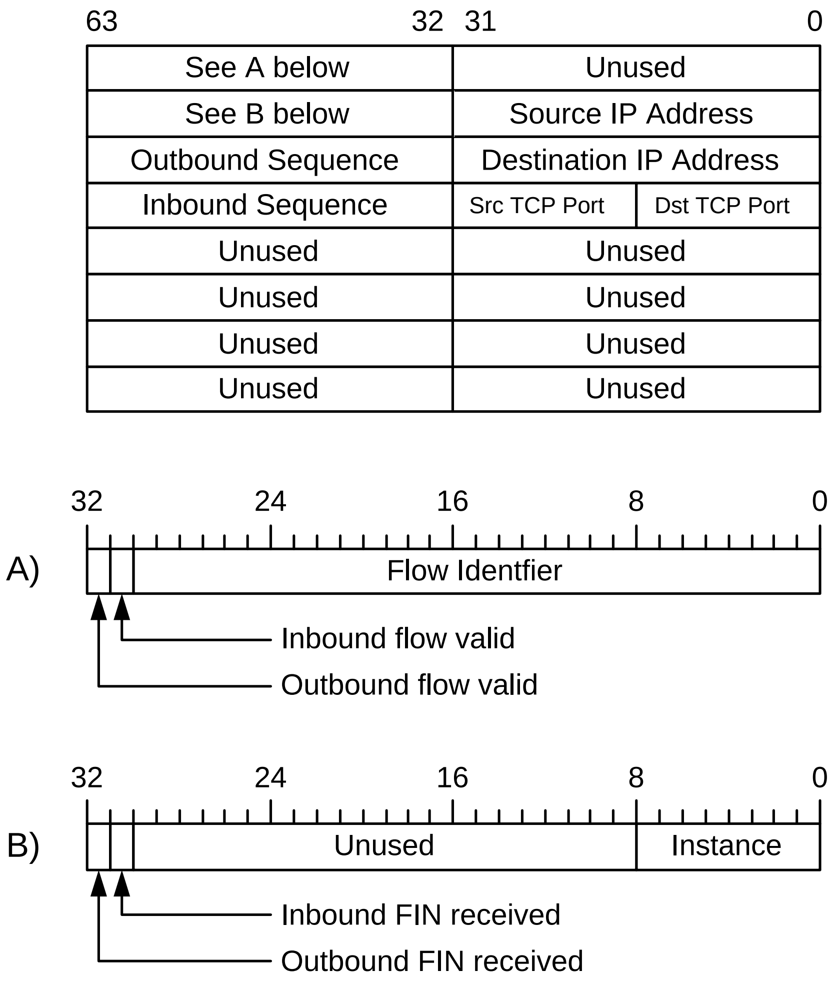


Figure 13: Per-Flow Record Layout

The current sequence number for both outbound and inbound traffic and an 8-bit instance identifier are also stored within the per-flow record. This instance value is incremented each time a new flow is stored in a context record. Four 1-bit flags indicate whether or not outbound traffic is valid, inbound traffic is valid, outbound traffic has ended and inbound traffic has ended.

The next couple of processes indicate whether or not the frontside engine is active and if active, the memory address of the per-flow record being accessed. When both the frontside and backside engines are

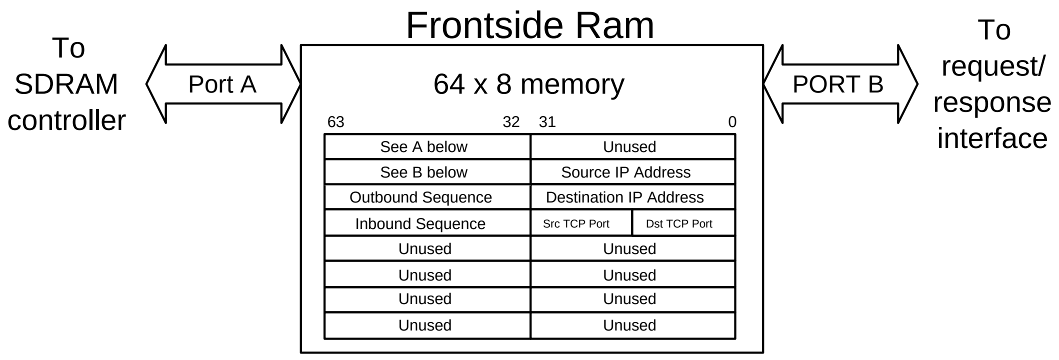


Figure 14: Frontside RAM Interface Connections

utilized, these signals are used to ensure that both engines don't try to manipulate the same context record, potentially leaving the record in an inconsistent state. In addition, three counters keep track of the take count, the app count, and the update count. The take count refers to the data words being returned from SDRAM. The app count refers to the application-specific context information stored in the context record. The storage of application specific context information is not utilized in this version of the TCP-Processor. The update count keeps track of memory update processing.

The SDRAM controller utilized by the StateStoreMgr contains a 64-bit wide data bus. The external request/response interface uses a 32-bit wide data bus. An internal dual-ported memory block moves data between these two busses. This memory block is called frontside_ram. Two engines control the movement of data in and out of the memory. As data is clocked in from external memory, the frontside_ram_A_engine writes this data into the frontside_ram. This data is written 64 bits at a time which matches the width of the memory interface. The frontside_ram_A_engine also reads data out of the frontside_ram where it is written to the external memory device. If the application context storage feature were enabled, then read operations utilizing the frontside_ram_B_engine would read application data from the frontside_ram to be used in the response sequence to the TCPEngine. The logic still exists for this task, but the state which drives the operation is never entered. Updates to the per-flow context record are written to the frontside_ram utilizing the frontside_ram_B_engine. Figure 14 shows the various interactions with the frontside_ram.

The following section of logic contains the write state machine which manages interactions with the write interface of the memory controller. This state machine is also responsible for initializing memory after reset. A series of states are entered which request the memory bus and write zeros to all memory locations of the external memory module. This initializes memory to a known state prior to any per-flow context record storage or retrieval operations. The write engine ensures the correct interaction with the write interface of the memory controller.

The next grouping of processes are responsible for incrementing a memory address utilized during memory initialization and a flag to indicate that initialization has completed. Additionally, there are signals to hold the update information passed in via the external interface. These signals include the new sequence number and a finish signal which indicates that the TCP connection is being terminated. The final process of the frontside section maintains a counter utilized when writing data to external memory.

The processes associated with the backside request/response interface are listed in the remainder of the source file. The actions by these processes closely mirror the actions of the frontside interface. Because of the similarity to the frontside processing and the fact that the backside interface is not used in this implementation of the TCP-Processor, detail will not be presented on each of these processes and signals.

3.11 TCP Routing

The TCPRouting component is responsible for routing the packets and associated information that arrive from the TCPEngine component and is implemented in the `tcprouting.vhd` source file. These packets can be routed to either the client monitoring interface (the normal traffic flow), the TCPEgress component (the bypass path), or the bit bucket (i.e. dropped from the network). Figure 15 shows the layout of the TCPRouting component.

The first process in the `tcprouting.vhd` source file generates the statistics event signals passed to the TCPStats component. These signals keep track of the number of packets passed to each of the output interfaces. Additionally, there is a two-bit vector which keeps an accurate count of the TCP stream bytes passed to the application interface. This count does not include retransmitted data. The next group of processes contain flops which copy all inbound data to internal signals. These internal signals are used when passing data to the output interfaces. One state machine in this component tracks the start and the end of a packet. The processing states include idle, data, length, and crc states. The final two states indicate the end of an IP packet. Since the TCPEngine contains a store and forward cycle, packets are always received by the TCPRouting component as a continuous stream of data.

The next two processes determine whether or not to enable the client application interface, the bypass

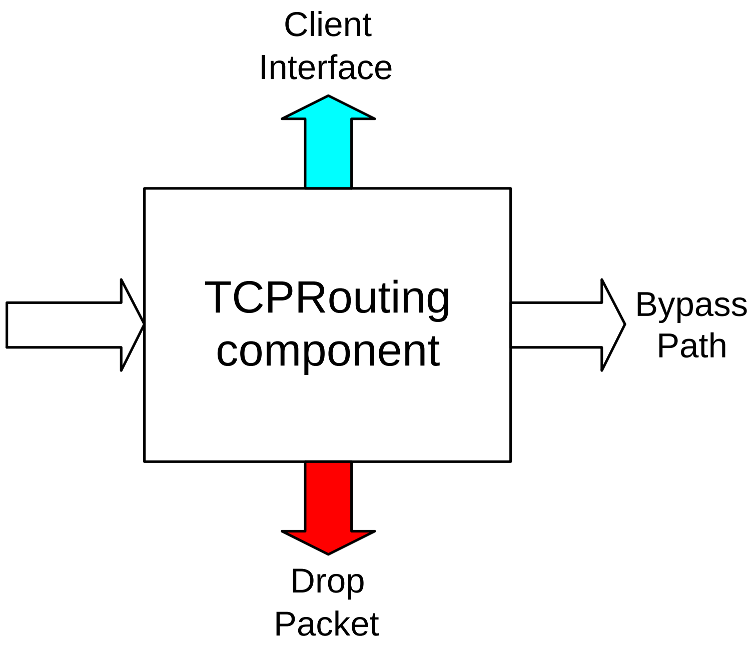


Figure 15: TCPRouting Component

interface, or neither interface (i.e. drop packet). Information passed in from the TCPEngine and the current configuration setting are used to make this decision. Two enable signals are generated which are used by subsequent processes which actually drive the output signals. The remainder of the processes drive output signals to either the client application interface or the bypass interface. These processes look at the current processing state and the value of the aforementioned enable signals to determine whether or not to connect the internal data signals to the output interfaces.

Figure 16 illustrates a waveform showing the interface signals from the TCP-Processor to the client TCP flow monitoring application. The interface consists of 12 output signals (data, dataen, sof, soip, sop, eof, tde, bytes, newflow, endflow, flowstate, and flowstaten) and one input signal (tca) used for flow control purposes. When the sof signal goes high, the processing of packet data is initiated. In conjunction with this signal, the newflow and/or endflow signals are driven indicating whether this packet is the start of a new flow or the end of an existing flow. It is possible for both the newflow and endflow signals to be high for the same packet. This can occur if the first packet processed for a particular flow has the RST or FIN flags set. The

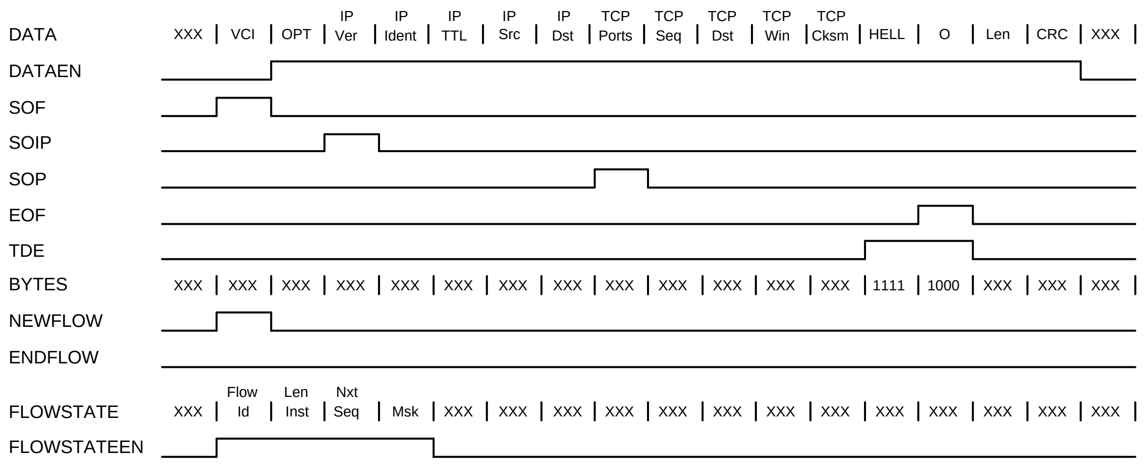


Figure 16: TCP Outbound Client Interface

flowstate and flowstaten signals are also driven in conjunction with the sof signal. The data bus contains the 32-bit ATM header (minus the HEC) associated with the packet. The dataen signal is low for the first word of the packet, but is driven high for all successive data words.

Following the initial clock cycle of the packet, there may exist zero or more lower layer protocol header words inserted prior to the start of the IP header. In this example, there is one such header word labelled OPT. The soip signal indicates the start of the IP header. Only IPv4 packets are supported, which means there will be at least 20 bytes of IP header data. If IP header options are present, they occur prior to the sop signal. The sop signal indicates the start of the IP payload. In this example, the sop signal corresponds to the start of the TCP header. The TCP header is 20 bytes or larger, depending on the number of included TCP options.

The packet in Figure 16 contains the five character text string *HELLO* in the TCP payload section of the packet. The client application processing the TCP stream content should utilize the tde and bytes signals to determine which data bytes to process as stream content. The tde (TCP data enable) signal indicates that the information on the data bus is TCP stream data. The four-bit-wide bytes vector indicates which of the four data bytes are to be considered part of the TCP data stream for that flow. When retransmissions occur, it is possible that the bytes vector indicates that none of the data bytes pertain to TCP stream data.

The end of the packet is indicated by the eof signal. Upon detection of a high eof signal, there are two

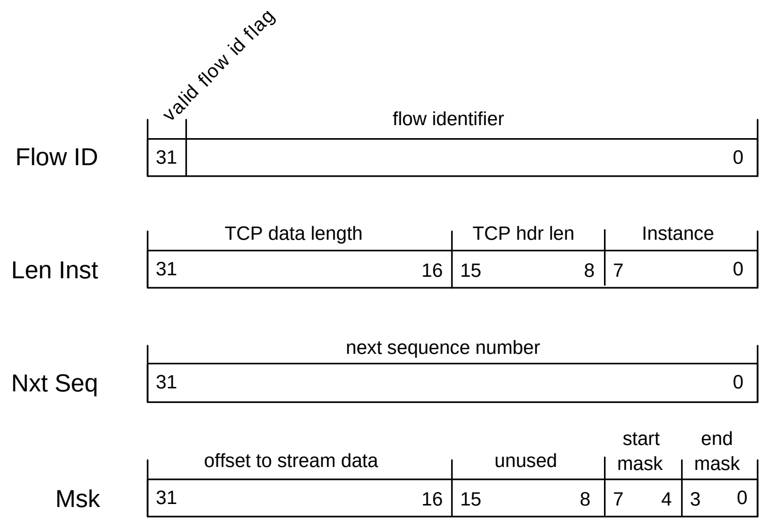


Figure 17: Flowstate Information

subsequent trailer words associated with the current packet which need to be processed. These trailer words contain AAL5 length and CRC information.

Every TCP packet provides four words of flowstate data. Figure 17 illustrates the exact format of these data bytes. The flow identifier is always the first word of flowstate information. Bit 31 indicates that the flow identifier is valid. The next data word contains the TCP data length in bytes, the TCP header length in words, and a flow instance number that can be used to help differentiate multiple flows which map to the same flow identifier. The next word contains the next expected TCP sequence number for this flow. This value can be useful when performing flow blocking operations. The final word contains a word offset to the continuation of TCP stream data with respect to this flow along with byte masks for the first and last words of the flow.

3.12 TCP Egress

The TCPEgress component merges traffic from different sources into one unified packet stream which is passed to the outbound IP wrappers and is implemented in the `tcpegress.vhd` source file. Three separate FIFOs are utilized to buffer traffic from the different sources during the merge operation. Additional processing is performed on the inbound traffic from the TCP flow monitoring client to validate and possi-

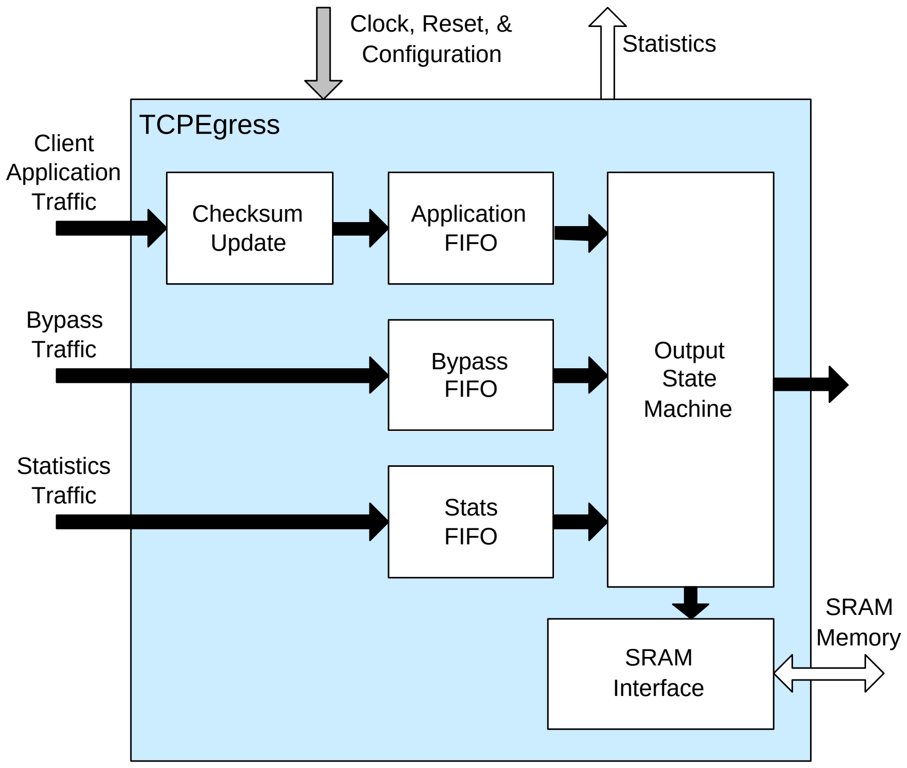


Figure 18: TCPEgress Component

bly update TCP checksum values. By recomputing the TCP checksum, the TCP-Processor supports client TCP processing circuits which modify TCP stream content. This can be accomplished by either inserting new packets (possibly a packet to terminate the TCP flow), or by modifying an existing packet. The IP-Wrapper supports special trailer words which indicate modifications that need to be made to the packet. The TCPEgress component does not take these changes into consideration when performing its checksum calculation. If the client uses this mechanism to modify the packet (either header or body), the calculated checksum value will be incorrect and the packet will eventually be dropped. To operate properly, the TCPEgress component must receive properly formatted TCP/IP packets (excluding the checksum values).

Like the previous components, the `tcpegress.vhd` source file begins with a process which generates statistics events which are passed to the TCPStats component. Stats are kept on the number of packets processed from the client interface, the number of packets processed from the bypass interface, and the

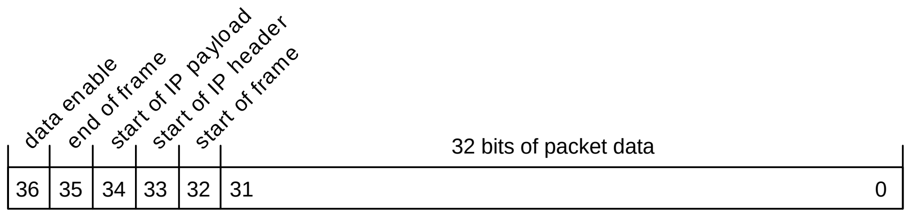


Figure 19: TCPEgress FIFO Format

number of TCP checksum update operations that were performed.

The next set of processes copy input signals from the stats and bypass interfaces and flop them into a set of internal signals. These internal signals are then used to write the packet contents to the stats and bypass FIFOs. The stats, app and bypass FIFOs all contain packet data in the same format. This format can be seen in Figure 19. There is also a set of processes which clock inbound packets from the client interface through five separate signals. This five cycle delay is utilized later so that checksum results can be made available before the end of the packet is reached. This group of logic is straightforward and no additional processing on the packets is performed.

The next section of logic performs the TCP checksum validation operation. An input state machine tracks progress through the various parts of the IP packet. This state machine has similar states and state transitions as the input state machine contained within the TCPEngine component (which also performs a TCP checksum validation operation). Signals are present which compute the length of the IP header and the data length of the TCP packet. This information is required to properly navigate the packet and compute the checksum value. Next is the set of processes which actually perform the checksum calculation. Since data is clocked in 32 bits at a time, checksum calculations are performed separately on the upper and lower 16 bits of the packet. The results of these separate checksum calculations are combined to produce the final checksum result. Following the checksum calculations, a series of processes hold the new checksum value, a determination as to whether the packet checksum is valid, the offset to the checksum value, and the checksum value contained in the packet.

The next group of processes is responsible for clocking packets from the client application interface into the application FIFO. The sequence of events is controlled by a simple state machine which determines whether the recomputed checksum value should be added to the end of the packet as an update command to the IPWrapper. If a checksum update is required, the state machine enters the Cksum and Delay states which supports inserting the update command into the packet trailer.

The next section of code contains the output state machine which is responsible for pulling data out of the three internal FIFOs and passing packets to the outbound IPWrapper interface. The state machine prioritizes the traffic in the three FIFOs such that packets are always retrieved from the stats FIFO first, the bypass FIFO second, and the app FIFO third. The last_read signal is used to ensure that a complete packet is read from the various FIFOs. Timing issues can arise when the state machine attempts to read the last word of a packet from a FIFO and the FIFO is empty because of the delay between when the read command is issued and when the data is returned from the FIFO. The last_read signal helps the processes deal with this situation in order to prevent deadlocks.

The final set of processes controls interactions with the SRAM memory module. Outbound packets are written to the SRAM device until it fills up with 256k words or packet data. This packet data can be retrieved from memory utilizing the SRAMDUMP utility. Packets are stored in the standard frame format as shown in Figure 8. All of SRAM is initialized to zero after receiving a reset command. The TCPEgress component will not start processing packets until the memory initialization is complete. Utilizing the *simulation* configuration parameter allows simulations to run quickly by skipping the memory initialization operation.

Packets passed from the client monitoring application to the TCP-Processor must be formatted in the same manner as packets passed to the client monitoring application. The major difference between the format of packets passed to the client and the fomrat of packets passed back from the client is that not all of the control signals are required for packets returned from the client. As seen in Figure 20 only the data, dataen, sof, soip, sop and eof signals are necessary. Other than dropping the remaining signals, the interface specification is identical to that of packets delivered through the outbound client interface of the TCP-Processor.

3.13 TCP Stats

The TCPStats component collects statistical information from other components, maintains internal event counters, and generates UDP-based packets on a periodic basis containing a summary of the event counts during the previous collection period. This logic is implemented in the `tcpstats.vhd` source file.

The first two processes in the `tcpstats.vhd` source file maintains the cycle counter and the collection interval trigger. The counter is a free running clock which is reset whenever the the interval period (defined

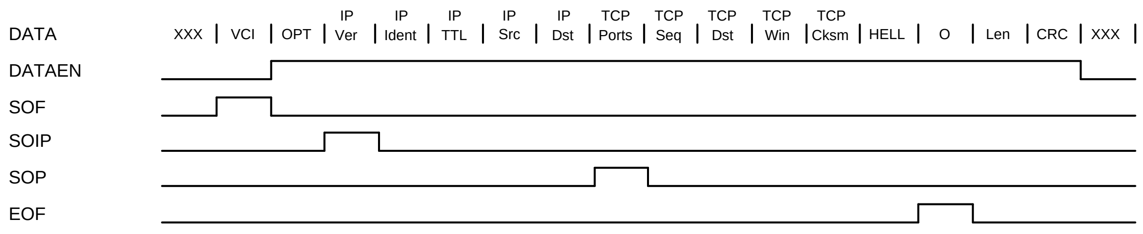


Figure 20: TCP Inbound Client Interface

by configuration parameter) is reached. The counter, composed of four separate 8-bit counters, eliminates the need for a single 32-bit-wide carry chain. The trigger fires when these four counter values match the state_cycle_counter configuration parameter.

The next group of processes contain 16, 24 and 32-bit event counters. A separate counter is maintained for each statistics variable maintained by the TCPStats component. These counters are reset when the trigger fires, indicating the end of the collection period.

The packet state machine is responsible for cycling through the processing states required to generate a UDP packet containing a summary of the statistics information gathered over the previous collection period. The trigger signal initiates the generation of the statistics packet. Once started, the process cannot be stopped and the state machine traverses through all of the states required to send the packet.

The next process is responsible for storing the value of each statistic counter in a temporary holding location while the statistics packet is being generated. When the trigger fires, signalling the end of the collection period, all of the current counter values are saved before being reset to zero. This provides a consistent snapshot of all the counters in a single clock cycle.

The next process is responsible for generating the outbound UDP-based statistics packets. Information on the packet formats and data collection tools can be found at the following URL:

`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp`
`/statscollector.html`

Figure 21 shows the processing states and the associated packet information relating to the statistics packet. To add additional statistics to this packet, the developer will have to modify the UDP packet length (the high 16 bits of the Cksum word), the stats count (the low 8 bits of the Hdr word), and insert the new statistics parameters at the end of the current parameters.

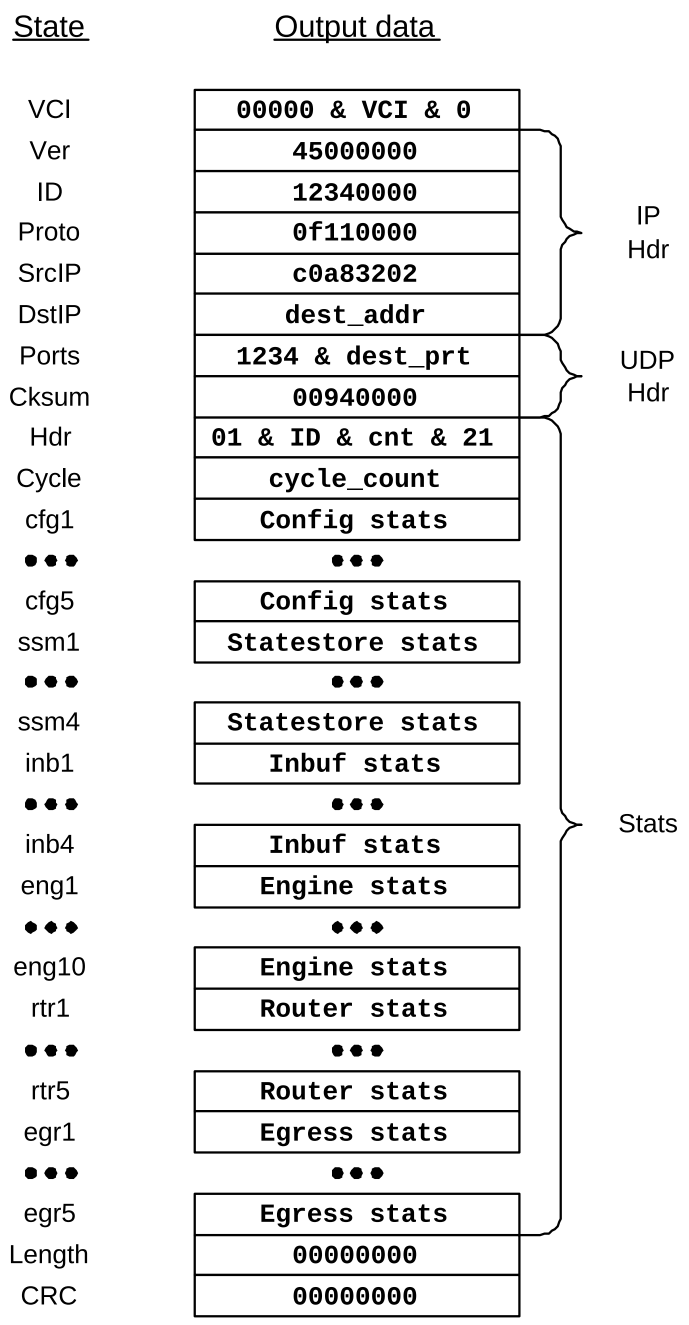


Figure 21: Stats Packet Format

The exact sequence of statistics values is listed below:

- ●cfg1: simulation
- ●cfg2: application bypass enabled
- ●cgf3: classify empty packets
- ●cfg4: checksum updated enabled
- ●cfg5: skip sequence gaps
- ●ssm1: new connections
- ●ssm2: end (terminated) connections
- ●ssm3: reused connections
- ●ssm4: active connections
- ●inb1: inbound words
- ●inb2: inbound packets
- ●inb3: outbound packets
- ●inb4: dropped packets
- ●eng1: inbound packets
- ●eng2: TCP packets
- ●eng3: TCP SYN packets
- ●eng4: TCP FIN packets
- ●eng5: TCP RST packets
- ●eng6: zero length TCP packets
- ●eng7: retransmitted TCP packets
- ●eng8: out-of-sequence TCP packets
- ●eng9: bad TCP checksum packets
- ●eng10: outbound packets
- ●rtr1: inbound packets
- ●rtr2: client outbound packets
- ●rtr3: bypass outbound packets
- ●rtr4: dropped packets
- ●rtr5: TCP data bytes
- ●egr1: inbound client packets

- egr2: outbound client packets
- egr3: inbound bypass packets
- egr4: outbound bypass packets
- egr5: TCP checksum updates

4 StreamExtract

The StreamExtract circuit provides primary processing of network packets in multi-board TCP flow monitoring solutions. As the name implies, TCP byte streams are *extracted* from the network traffic and passed to a separate device for additional processing. Network traffic enters the circuit through the RAD Switch interface and is processed by the Control Cell Processor (CCP) and the TCP-Processor. The StreamExtract circuit connects to the TCP processor as the client TCP flow monitoring application. This component encodes/serializes the data provided through the client interface and passes it out through the RAD Line Card interface. This traffic can then be routed to other hardware devices which perform TCP stream processing. Data is returned via the RAD Line Card interface in this same format to the StreamExtract circuit where it is decoded/deserialized and returned to the TCP-Processor. After egress processing by the TCP-Processor, network traffic is passed to the RAD Switch interface. Additional information regarding the StreamExtract circuit can be found at the following web site: `http://www.arl.wustl.edu/projects/fpx /fpx_internal/tcp/stream_extract.html`

Figure 22 shows the layout of the StreamExtract circuit. The top level rad_streamextract entity interfaces directly to the I/O pins on the FPX platform, the VHDL code for which is located in the `rad_stream-extract.vhd` source file. All external signals are brought through this interface. A one clock cycle buffer delay is added to all of the UTOPIA I/O signals which carry network traffic. The sram_req_fixup process ensures that the CCP module has priority access to the dual ported SRAM memory interface. This means that external access will be available to memory, even when another component locks up while controlling the memory device.

The remainder of the source is utilized to glue the various components together. The SDRAM sequencer and SRAM interface memory controllers provide multiple interfaces and simplify access to the external memory devices. All of the I/O signals are then passed into the StreamExtract_module.

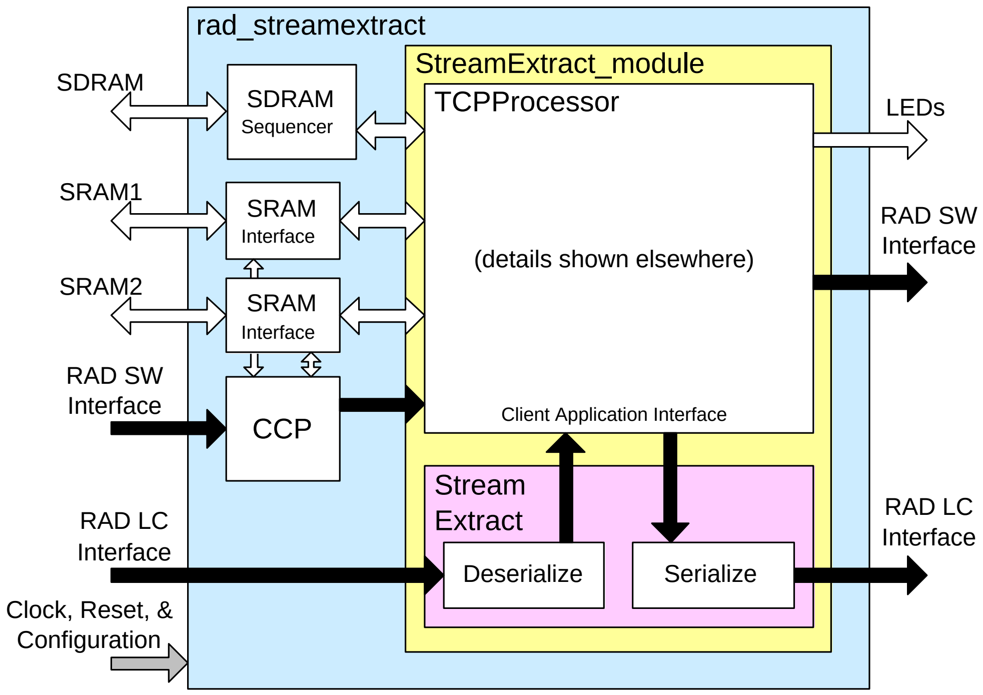


Figure 22: StreamExtract Circuit Layout

4.1 StreamExtract_Module

The StreamExtract_module component encapsulates the interaction between the TCPProcessor and the StreamExtract components. Both the TCPProcessor and the StreamExtract components contain two SRAM interfaces which can be used to capture network traffic for debugging purposes. Since there are only two SRAM devices on the chip, the StreamExtract_module contains two NOMEM dummy memory interfaces which connect to two of the four memory interfaces. These NOMEM interfaces act like memory devices which are always available, but the data is never written anywhere and read operations always return the value zero.

The `streamextract_module.vhd` source file also contains all of the configuration information for the circuit. After modifying the configuration signal values in this file, rebuild the circuit in order for the changes to be incorporated into it. The specific function of each configuration parameter and possible values are described in the *Configuration Parameters* section.

4.2 StreamExtract

The StreamExtract component connects the TCPSerializeEncode and TCPSerializeDecode components to the external StreamExtract interfaces. These components encode the client interface of the TCP-Processor and enable transport to other devices.

4.3 LEDs

The four RAD controlled LEDs on the FPX platform provide a limited amount of information about the operation of the StreamExtract circuit. LED1 is connected to a timer which causes the LED to blink continuously once the circuit has been loaded. The rest of the LEDs indicate the current status of various flow control signals. LED2 is illuminated when the flow control signal from the TCPEngine to the TCPInbuf component is low. LED3 is illuminated when the flow control signal from the TCPRouting to the TCPEngine component is low. LED4 is illuminated when the flow control signal from the outbound IPWrapper to the TCPEgress module is low. During normal operation, these three LEDs should be off, or at most only flash for a very brief time period. A constantly lit LED indicates that a portion of the circuit is locked-up and is not passing data.

4.4 Serialization/Deserialization (Endoding/Decoding)

The TCPSerializeEncode component processes data from the outbound client interface of the TCP-Processor. The information is encoded in a self-describing and extensible manner which supports transmitting the information to other devices. Likewise, the TCPSerializeDecode component receives the encoded information and regenerates the signals required by the inbound client interface of the TCP-Processor. Figure 23 shows the mechanics of the encoding and decoding operations. The extra information provided along with the packet data in the client interface signals from the TCP-Processor is compressed and grouped into several control headers which are prepended to the network packet. This information is then divided into ATM cells and transmitted through 32-bit UTOPIA interface to other devices. The decode operation performs the reverse operation, taking a sequence of ATM cells and reproducing an interface waveform identical to the one generated by the TCP-Processor.

The control header formats are described in detail utilizing standard C header file constructs. Every control header starts with an 8-bit `CTL_HEADER` which describes the type and length of the control header,

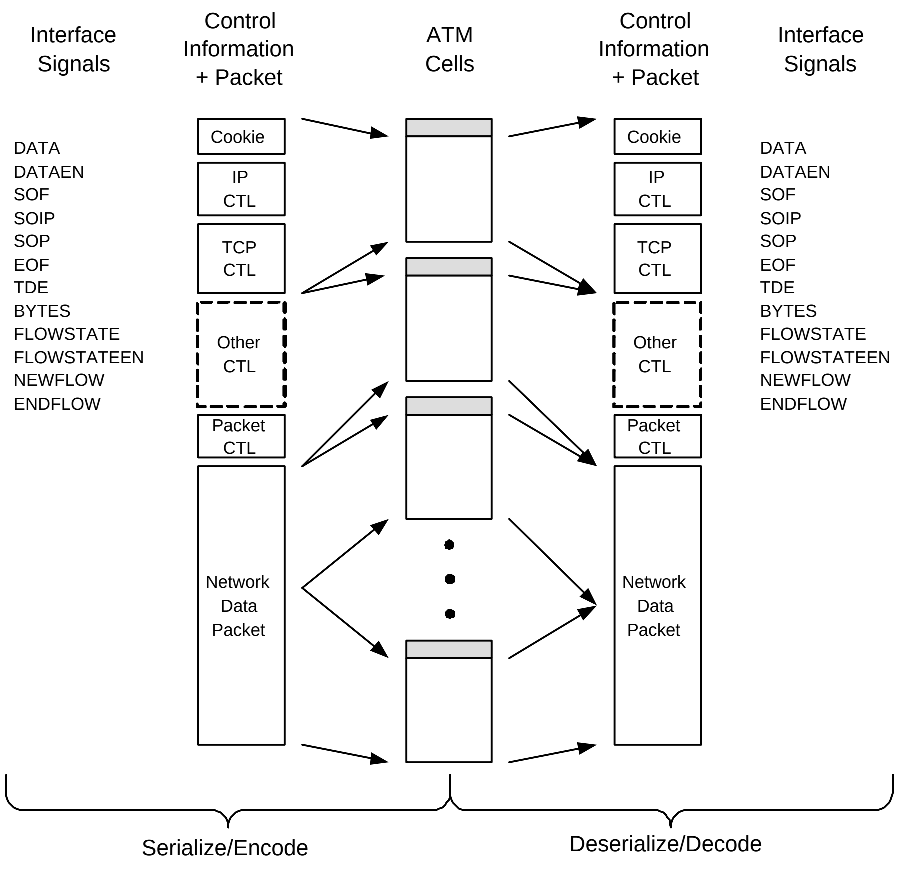


Figure 23: Packet Serialization and Deserialization Technique

the layout of which is shown in Figure 24. By utilizing a self-describing format for the control header, additional control headers can be defined and added in the future without requiring a rewrite of existing circuits. Because of this common header format, current circuits can easily bypass control headers they don't understand. This preserves the investment made in circuits today while enabling future enhancements. The C style coding of the currently defined header types is shown below.

```
/* Control Header Type Definitions */

#define CTL_TYPE_UNUSED        0
#define CTL_TYPE_IP_HEADER     1
#define CTL_TYPE_TCP_HEADER    2
```

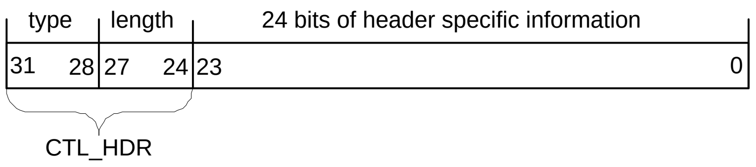


Figure 24: Control Header Format

```
#define CTL_TYPE_PAYLOAD       3
#define CTL_TYPE_COOKIE        4
/* other headers types */

#define CTL_TYPE_RESERVED     15

struct {
    unsigned char Type   :4; /* Type code */
    unsigned char Length :4; /* Header length */
                             /*   (number of words to follow) */
} CTL_HEADER;
```

Every packet encoded by these routines is an IP packet, since only IP packets are passed to the TCP-Processor. All encoded packets have the same sequence of control headers after the encoding process. This sequence includes a cookie control header (which acts as a record mark), an IP control header, possibly a TCP control header (depending on whether or not this is a TCP packet), and finally a packet control header. The layout of each of these headers is listed below:

```
/* cookie control header */
#define CTL_COOKIE    0x40ABACAB


/* IP control header */
struct {
/* 1st clock */
    unsigned int  HdrType    : 4;   /* standard header (Hdr.Length = 0) */
    unsigned int  HdrLength  : 4;   /* SOF is always high for the first word */
                                    /*   of the payload */
                                    /* DataEn is always high for the 2nd through */
                                    /*   the nth word of the payload */
    unsigned int  SOIPOffset : 6;   /* # of words past SOF to assert SOIP signal */
                                    /* SOIP is high for one clock cycle */
    unsigned int  SOPOffset  : 4;   /* # of words past SOIP to assert SOP signal */
                                    /* SOP remains high until the EOF signal */
    unsigned int  EOFOffset  : 14;  /* # of words past SOP to assert EOF */


} CTL_IP_HDR;


/* TCP control header */
struct {
/* 1st clock */
    CTL_HEADER Hdr;                 /* standard header (Hdr.Length = 3 or 0 */
                                    /*   depending on value of ValidFlow) */
    unsigned char spare1     : 3;   /* spare */
    unsigned char ValidFlow  : 1;   /* Indicates whether or not the flow ID */
                                    /*   has been computed for this packet */
                                    /* if this value is zero, then the */
                                    /*   Hdr.Length field is set to 0 and no TCP */
                                    /* control information follows this header */
    unsigned char TDEOffset   : 4;  /* # of words past SOP to assert TDE signal */
                                    /* TDE remains high until the EOF signal */
    unsigned char StartBytes  : 4;  /* Valid bytes at start */
    unsigned char EndBytes    : 4;  /* Valid bytes at end */
    unsigned char Instance;         /* Flow instance */

/* 2nd clock */
    unsigned short StartOffset;     /* Offset from TDE to where to start */
                                    /*   supplying ValidBytes data */
    unsigned short TCPDataLength;   /* Number of TCP data bytes contained */
                                    /*   within packet */

/* 3rd clock */
    unsigned int NewFlow      : 1;  /* Indicates that this is the start of a */
                                    /*   new flow */
    unsigned int EndFlow      : 1;  /* Indicates that this is the end of an */
                                    /*   existing flow */
    unsigned int FlowID       :30;  /* Flow Identifier */

/* 4th clock */
    unsigned int NextSequence;      /* Next sequence identifier */

} CTL_TCP_HDR;


/* packet control header */
struct {
/* 1st clock */
    CTL_HEADER Hdr;                 /* standard header (Hdr.Length = 0) */
    unsigned char spare;            /* spare */
```

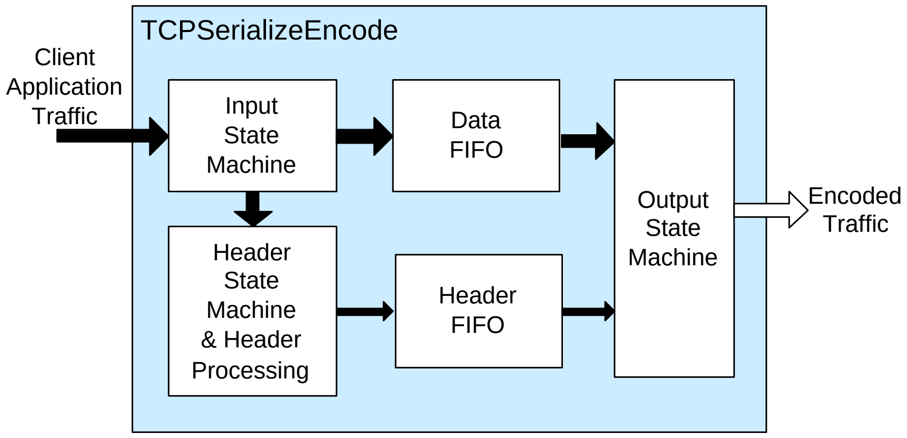


Figure 25: TCPSerializeEncode Circuit Layout

```
    unsigned short DataLength;      /* Number of data words to follow */
/* 2nd -> nth clock */              /* payload data words */
} CTL_PAYLOAD;
```

4.4.1 TCPSerializeEncode

The `tcpserializeencode.vhd` source file contains all of the circuit logic required to encode signals from the TCP-Processor client interface into a protocol which can easily be transported to other devices. The operation of this circuit is driven by three separate state machines: an input state machine, a header state machine, and an output state machine, as seen in Figure 25.

The first several processes in the source file contain various counters which are commented out. These counters were utilized during debugging and are not required for correct operation of the circuit. The first active process in the source file copies the signals from the TCP-Processor client interface to internal signals. The input state machine tracks the beginning and the end of every packet. This state information is utilized by the following process to insert packet data into the data FIFO. Because the TCP-Processor contains a store and forward cycle, there are no gaps in the in the data. More specifically, all words of a packet will arrive on consecutive clock cycles. Because there are no gaps, the input data processing does not need to watch the data enable signal for detecting valid data.

The next series of processes extract or compute information necessary to produce the control headers of the encoded data format. This information includes offsets from the start of the packet to the IP header, the IP payload, the end of packet, and the TCP payload data. In addition, data delivered via the flowstate bus is stored. The header state machine drives the generation of the flowstate headers from the information collected and computed by the previous processes. Following the state machine is the control logic which inserts the control headers into the header FIFO.

The output state machine is responsible for generating the outbound ATM cells of the encoded format. Processing is initiated when a non-empty header FIFO is detected. The encoding format assumes that no cells will be lost during transmission between multiple devices, so ATM adaptation layer zero (AAL0) cells are generated. These ATM cells are first filled with header information read out of the header FIFO and then with packet data read out of the data FIFO.

The final set of processes contains logic for saving the encoded packet data contained in ATM cells to external SRAM. This feature is available to aid in debugging encoding, decoding or data processing problems. All of SRAM is initialized to zero prior to packet processing. The information stored in SRAM should be identical to the encoded data transmitted out the RAD Line Card interface by the StreamExtract circuit.

4.4.2 TCPSerializeDecode

The logic for the TCPSerializeDecode circuit is contained in the `tcpserializedecode.vhd` source file. This circuit takes packets which have been encoded into ATM cells and reproduces the waveform required by the TCPEgress component of the TCP-Processor. Figure 26 illustrates the basic layout of this circuit.

The first process in this source file copies the inbound interface signals into internal signals used for the rest of the data processing. The cell state machine is responsible for processing every ATM cell, differentiating between the cell header and the cell payload. The cell payload is passed downstream for additional processing.

The input state machine is responsible for tracking the progress through the encoded control headers and the packet payload. To initiate processing, the cookie control header must be detected as the first word of an ATM packet. If the cookie is detected, the state machine enters the header processing state. Transition out of this state into the payload state occurs when the payload control header is detected.

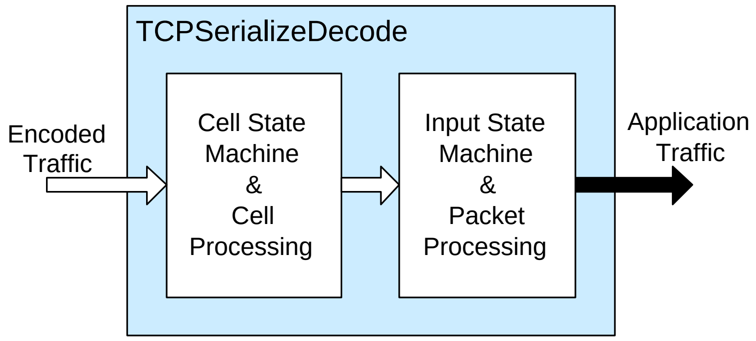

TCPSerializeDecode
Encoded Traffic
Cell State Machine & Cell Processing
Input State Machine & Packet Processing
Application Traffic

Figure 26: TCPSerializeDecode Circuit Layout

The next group of processes are responsible for extracting information out of the control headers and generating the associated packet control signals, such as the start of IP header, the start of IP payload, the start of frame, the end of frame, and the data enable signal. These output signals are produced by counting down a packet offset count until the appropriate signal should be generated.

As with the TCPSerializeEncode circuit, the TCPSerializeDecode circuit contains logic to write inbound ATM cells to SRAM memory. All of SRAM is initialized to zero prior to packet processing. The information stored in SRAM should be identical to the encoded data received via the RAD Line Card interface by the StreamExtract circuit.

5 TCPLiteWrappers

The TCPLiteWrappers[1] provide an environment in which TCP flow processing applications can operate. They are composed of the TCPDeserialize and TCPReserialize circuits. The TCPDeserialize circuit takes the encoded data stream generated by the TCPSerializeEncode circuit and converts it into the original client interface waveform produced by the TCP-Processor. The TCPReserialize circuit takes the output of the client TCP processing circuit and re-encodes the data for transmission to other devices. The output format of the TCPDeserialize circuit is identical to the input format of the TCPReserialize circuit. Likewise, the input format of the TCPDeserialize circuit is identical to the output format of the TCPReserialize circuit. Given the commonality of these interfaces, the pair of the TCPDeserialize and TCPReserialize circuits have

[1]The TCPLiteWrappers moniker was created by James Moscola. Previously, the TCPLiteWrappers were unnamed.

a net zero effect on data that they process and could be inserted multiple times into a given application.

The source for the TCPDeserialize and TCPReserialize circuits can be found in either the distribution for *PortTracker* or *Scan*. The source files are identical in both distributions.

5.1 TCPDeserialize

The TCPDeserialize circuit is different from the TCPSerializeDecode circuit because the TCPSerializeDecode circuit only needs to reproduce a subset of the client interface signals as required by the inbound TCPEgress component. The TCPDeserialize component needs to faithfully reproduce all of the original TCP-Processor client interface signals. The minor differences between the two circuits should be easy to comprehend. The `tcpdeserialize.vhd` source file contains the source code for the TCPDeserialize circuit.

5.2 TCPReserialize

Although similar, the TCPReserialize circuit is different from the TCPSerializeEncode circuit because the TCPSerializeEncode circuit can assume that all packet data will be received in consecutive clock cycles. This condition does not hold true for the TCPReserialize circuit which contains extra logic to deal with gaps in packet transmission.

The `tcpreserialize.vhd` source file contains the source code for the TCPReserialize circuit. As stated previously, the circuit layout, behavior, processes and signals for this circuit are almost identical to those of the TCPSerializeEncode circuit. Please refer to the documentation on the TCPSerializeEncode circuit in order to understand the operation of the TCPReserialize circuit.

5.3 PortTracker

The PortTracker application processes TCP packets and maintains counts of the number of packets which are sent to or received from selected TCP ports. Utilizing this circuit, one can understand the distribution of packets amongst the various TCP ports. This circuit also provides an excellent example of how to utilize the TCPLiteWrappers when the client TCP stream processing application is not concerned about maintaining state information on a per-flow basis. The source for the PortTracker application can be found at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal`

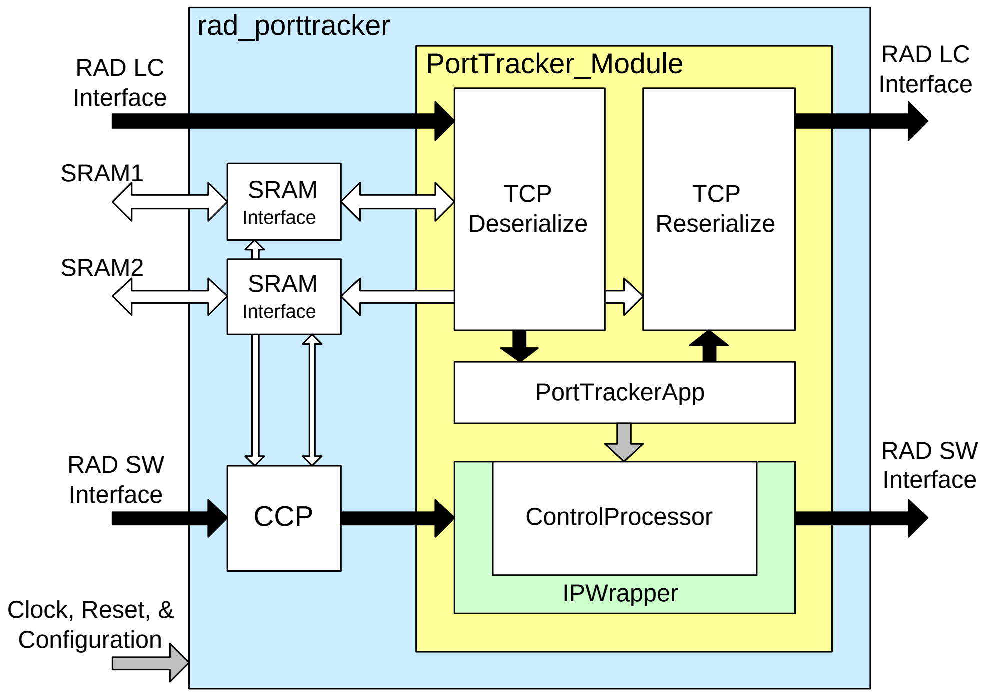


Figure 27: PortTracker Circuit Layout

`/tcp/source/porttracker_source.zip` The directory structure layout, circuit layout, and build procedures for this application are similar to that of the StreamExtract circuit.

Figure 27 shows a layout of the components which make up the PortTracker application. The main data flow is shown along with interactions with SRAM devices which capture traffic to aid in the debugging process. The *rad_porttracker* component is the top level design file which interfaces directly to the I/O pins of the RAD. The VHDL code for this component can be found in the `rad_porttracker.vhd` source file. Control cells and normal network packets are passed into the circuit via the RAD Switch interface. This data is first passed to the Control Cell Processor (CCP) which allows remote access to the SRAM devices. Encoded TCP stream data enters the circuit via the RAD Line Card interface. Network data from both of the external interfaces is passed into the PortTracker_Module for processing. The SRAM Interface components expose multiple memory access ports which allow for request multiplexing so that multiple components can share the same memory device.

5.3.1 PortTracker_Module

The VHDL code for the PortTracker_Module can be found in the `porttracker_module.vhd` source file. This component acts as the glue which holds all the subcomponents of the PortTracker circuit together and provides data routing and signal connections between the subcomponents and the external interface. Data from the RAD Switch interface is passed to the IPWrapper which converts the inbound cells into IP packets which are then passed to the ControlProcessor. Outbound traffic from the ControlProcessor is passed back to the IPWrapper which converts packets back into ATM cells for transmission out the RAD Switch interface. Encapsulated TCP stream data enters through the RAD Line Card interface and is passed to the TCPDeserialize circuit. From there, annotated TCP packets are passed to the PortTrackerApp circuit which scans network traffic for packets sent to or received from various TCP ports. Network traffic is then routed to the TCPReserialize circuit which re-encodes the network packets for transmission out the RAD Line Card interface.

The PortTracker_Module also contains the configuration parameters which define the operation of the PortTracker application. These configuration parameters have identical names and functions as those described in the StreamExtract circuit. Please refer to the previous ConfigurationParameters subsection in the Internals section of this document for details on each of the configuration parameters.

5.3.2 PortTrackerApp

The implementation for the PortTrackerApp can be found in the `porttracker.vhd` source file. The PortTracker application is a very simple circuit and is therefore a good example of how to implement a basic TCP packet processing application. The first process copies the inbound data into internal signals which are used in subsequent processes. The *istcp* signal indicates whether or not the current packet is a TCP packet. The *hdr_count* signal is a remnant from previous logic and is not used in the application. The *port_flop* process performs all of the work for the PortTracker application. The *SOP* and *istcp* signals indicate the presence of the TCP port number on the data bus. When this situation occurs, the source and destination TCP ports are run through a series of *if* statements which determine the type of packet being processed. Based on the result of the port comparisons, a port type signal is driven. Packets are always associated with the lowest port match (either source or destination) and packets are only counted once. The various port type signals are passed to the ControlProcessor for collection and generation of statistics packets.

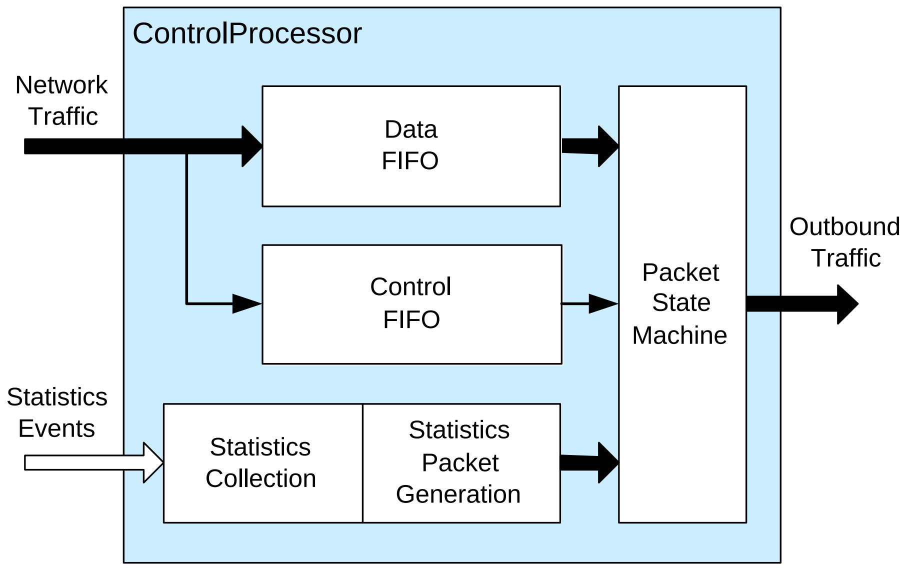


Figure 28: PortTracker ControlProcessor Layout

The following group of processes is responsible for copying the internal packet data signals to the external interface. Additionally, there are a couple of processes which flash the RAD LEDs. These LEDs have no meaning other than to indicate that the circuit has been loaded into the FPGA on the FPX platform.

5.3.3 ControlProcessor

The ControlProcessor component is described in the `control_proc.vhd` source file and overall flow of data is shown in Figure 28. The first several processes copy inbound network data into internal signals and load the packets into the Frame FIFO. Once a complete packet has been written to the Frame FIFO, one bit is written to the control FIFO to indicate that a complete packet is ready for outbound transmission.

The next couple of processes are responsible for maintaining the interval timer which drives the generation of the trigger event. This operation is very similar to that of the TCPStats module of the TCP-Processor. The trigger event initiates the saving of all statistics counters and the transmission of a statistics packet. The next group of processes are responsible for maintaining statistics counters for each of the port ranges which are tracked by this circuit. Upon reception of the trigger event, these counters are saved and reset.

The packet state machine is responsible for managing the transmission of outbound traffic. Upon detection of the trigger event, the state machine sequences through a set of states which are responsible for generating the statistics packet and sending it out via the outbound interface. When a non-empty condition is detected for the control FIFO, a network packet is read from the frame FIFO and transmitted out the output interface.

The next process is responsible for taking a snapshot of all the statistic counters and saving the current values when the trigger event fires. This allows the counters to reset and continue counting while the statistics packet is being generated.

The format of the statistics packet follows the same format as previously mentioned in this document. The PortTracker application generates statistics packet with a stats identifier of five (5). The exact sequence of statistics values transmitted in the statistics packets is listed below:

- cfg1: simulation
- numFTP: number of FTP packets
- numSSH: number of SSH packets
- numTelnet: number of telnet packets
- numSMTP: number of SMTP packets
- numTIM: number of TIM packets
- numNameserv: number of Nameserv packets
- numWhois: number of Whois packets
- numLogin: number of Login packets
- numDNS: number of DNS packets
- numTFTP: number of TFTP packets
- numGopher: number of Gopher packets
- numFinger: number of Finger packets
- numHTTP: number of HTTP packets
- numPOP: number of POP packets
- numSFTP: number of SFTP packets
- numSQL: number of SQL packets
- numNNTP: number of NNTP packets
- numNetBIOS: number of NetBIOS packets
- numSNMP: number of SNMP packets

•numBGP: number of BGP packets

•numGACP: number of GACP packets

•numIRC: number of IRC packets

•numDLS: number of DLS packets

•numLDAP: number of LDAP packets

•numHTTPS: number of HTTPS packets

•numDHCP: number of DHCP packets

•numLower: number of packets to/from TCP ports $\geq$ 1024

•numUpper: number of packets to/from TCP ports $>$ 1024

5.4 Scan

The Scan application processes TCP data streams and searches for four separate digital signatures of up to 32 bytes in length. This circuit provides TCP stream content scanning features which support detection of signatures which cross packet boundaries. This circuit also provides an excellent example/model/starting point of how to utilize the TCPLiteWrappers when the client TCP stream processing application needs to maintain context information for each flow in the network. The source for the Scan application can be found at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal /tcp/source/scan_source.zip`. The directory structure layout, circuit layout, and build procedures for this application are similar to that of the StreamExtract and PortTracker circuits. Additional information regarding the Scan circuit can be found at the following web site: `http://www.arl.wustl.edu /projects/fpx/fpx_internal/tcp/scan.html`

Figure 29 shows a layout of the components which make up the Scan application. The main data flow is shown along with interactions with memory devices which capture traffic to aid in the debugging process and store per-flow context information. The *rad_scan* component is the top level design file which interfaces directly to the I/O pins of the RAD. The VHDL code for this component can be found in the `rad_scan.vhd` source file. Control cells and control packets are passed into the circuit via the RAD Switch interface. This data is first passed to the Control Cell Processor (CCP) which allows remote access to the SRAM devices. Encoded TCP stream data enters the circuit via the RAD Line Card interface. Network data from both of the external interfaces is passed into the Scan_Module for processing. The SRAM Interface component exposes multiple memory access ports to provide request multiplexing so that multiple components can share the

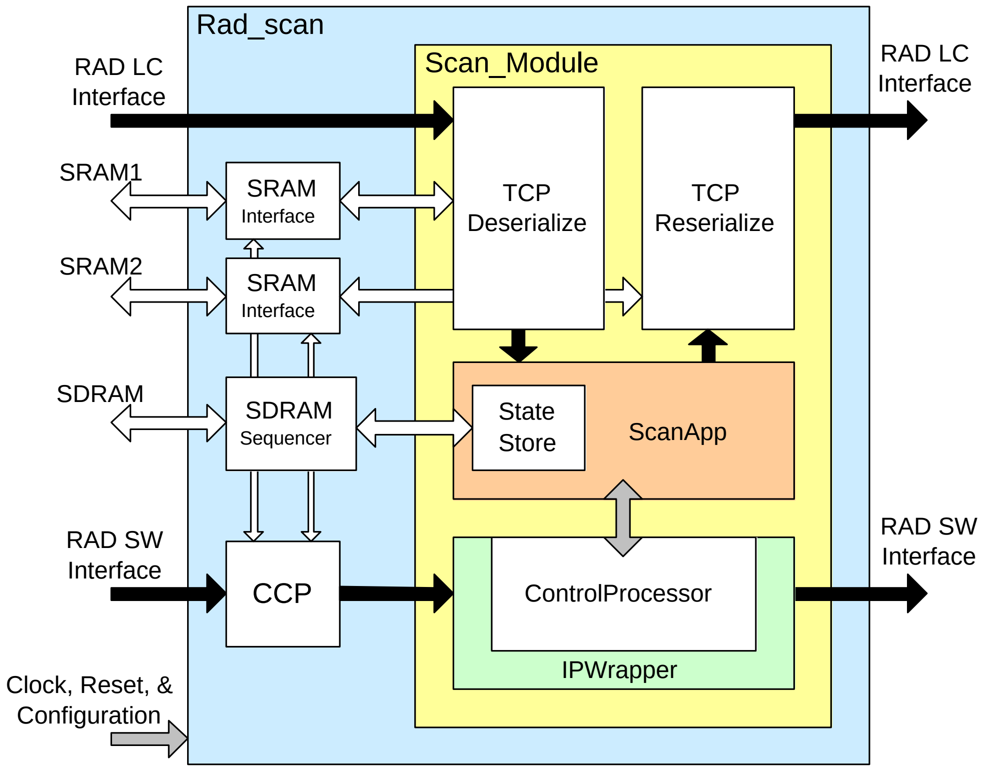


Figure 29: Scan Circuit Layout

same memory device.

5.4.1 Scan_Module

The VHDL code for the Scan_Module is in the `scan_module.vhd` source file. This component acts as the glue which holds all the subcomponents of the Scan circuit together and provides data routing and signal connections between the subcomponents and the external interface. Data from the RAD Switch interface is passed to the IPWrapper which converts the inbound cells into IP packets which are then passed to the ControlProcessor. Outbound traffic from the ControlProcessor is passed back to the IPWrapper which converts packets back into ATM cells for transmission out the RAD Switch interface. Encapsulated TCP stream data enters through the RAD Line Card interface and is passed to the TCPDeserialize circuit. From there, annotated TCP packets are passed to the ScanApp circuit which scans network traffic for four separate digital signatures. Network traffic is then routed to the TCPReserialize circuit which re-encodes the network packets for transmission out the RAD Line Card interface.

The Scan_Module also contains the configuration parameters which define the operation of the Scan application. These configuration parameters have identical names and functions as those described in the StreamExtract and PortTracker circuits. Please refer to the previous ConfigurationParameters subsection in the Internals section of this document for details on each of the configuration parameters.

5.4.2 ScanApp

The implementation for the ScanApp can be found in the `scan.vhd` source file. The internal logic layout of the ScanApp is shown in Figure 30. The main flow of network traffic goes left to right. The operation of the stream comparison logic is controlled by the output state machine. The most recent 32 bytes of stream data for each flow are stored in external memory via the StateStore component. This per-flow context information is loaded into the string comparison logic prior to processing a packet. After packet processing, this context information is saved to memory. There is also logic to control the replacement of search strings.

The first couple of processes perform component maintenance tasks. These tasks include controlling the TCA flow control signal for inbound data. The TCA signal is de-asserted whenever the data FIFO becomes 3/4 full or the control or header FIFOs become 1/2 full. Inbound data is also copied into internal signals which are used later in the circuit.

The input state machine manages the processing of inbound packets into the Scan circuit. Based on the input processing state, packet data is written into the data FIFO and flowstate information along with some control signals are written into the header FIFO. When the end of a packet is detected, a single bit is written to the control FIFO which indicates that a complete packet is available for downstream processing.

The next group of processes handles the processing of header information, issuing requests to the State-Store to retrieve per-flow context information, and processing the returned context information. The request state machine drives the sequence of operations required to interact with the StateStore. Upon detecting a non-empty header FIFO, the state machine progresses through states which retrieve flowstate information from the header FIFO. Depending on information retrieved from the header FIFO and the packet's relation to the previous packet (is this packet part of the same flow or not), the state machine will enter the WFR (wait for response) state, waiting for completion of the per-flow context retrieval from the StateStore. The WFO (wait for outbound processing) state is the last state entered, which ensures that data is held in internal

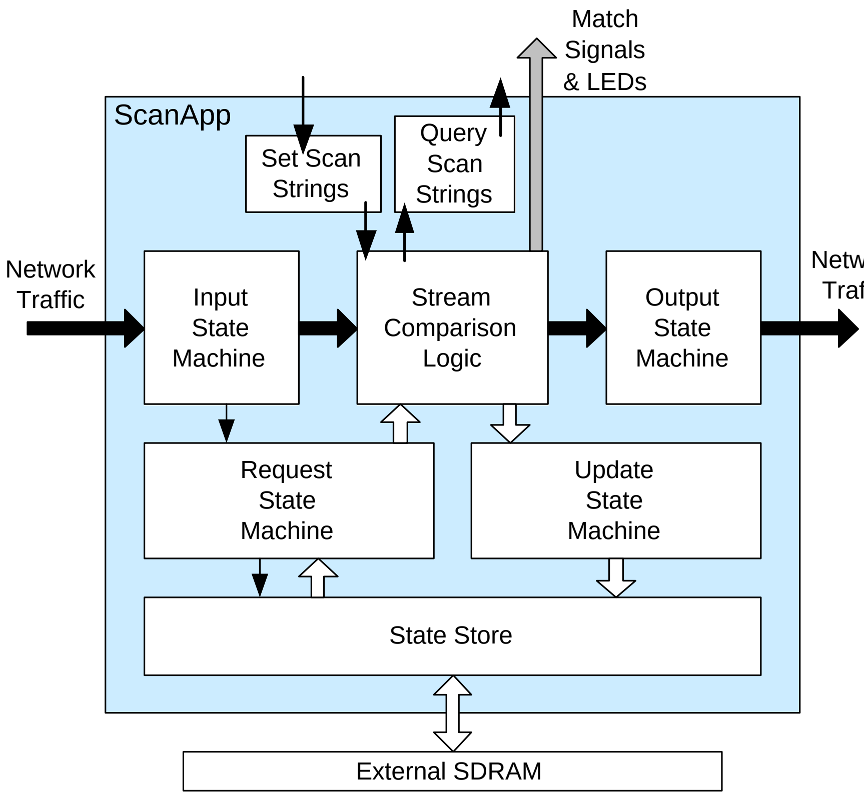


Figure 30: ScanApp Circuit Layout

buffers until outbound processing is initiated. The hfifo_control process controls the reading of information out of the header FIFO. The req_signals process is responsible for storing up to four words of flowstate information read from the header FIFO. The state_store_request process is responsible for driving the State-Store request signals which initiate the reading of per-flow context information. The retrieval_active flop indicates whether or not a retrieval operation of per-flow context information is active for this packet. Context information returned from the StateStore is saved in a set of signal vectors, prior to being loaded into the comparator.

The output state machine drives the processing of the stream comparison engine and the delivery of network traffic out of the component. When a non-empty condition is detected for the control FIFO and the request state machine is in the WFO state (indicating that StateStore request processing has completed for the next packet), the output state machine initiates outbound packet processing. A delay state is entered in order to wait for data to be clocked out of the data FIFO. There are special states (First, Second, Third and Fourth) for the first four words of a packet. This supports the special processing required to combine the flowstate information with the network packet. The Payload state manages the processing of the remainder of the packet. The Last1 and Last2 states handle processing of the AAL5 trailer which occurs at the end of the network packet. Following the state machine are a couple of processes which control the reading of data from the data and control FIFOs. The next process drives internal signals with packet data, flowstate data, and associated control signals.

The next three processes are used to store information required by the StateStore update logic after packet processing has been completed. Separate signals are used to store this information so that the front end processing (input and request state machines) can initiate processing of the next packet. These signals store the flow identifier associated with this packet, a determination of whether or not an update is required, and an indication of whether or not this is a new flow. If either the update required or the new flow signals are set, the update state machine will store the current flow context to memory via the StateStore component.

The next group of processes maintain the check string which contains the most recent 36 bytes of the TCP data stream for this flow. The valid_byte_count signal keeps track of how many new bytes of data (0, 1, 2, 3, or 4) are added to the check string. The build_check_string process is responsible for maintaining an accurate representation of the most recent 36 bytes of the TCP data stream. During processing of the first word of the packet, the check string is loaded with data either retrieved from the StateStore or initialized to zero. Subsequently, whenever there is valid TCP stream data on the data bus, the check string is updated accordingly. Since data is clocked in four bytes at a time, a 36-byte check buffer is maintained so that the four 32-byte comparisons can be performed on each clock cycle, each comparison shifted by one byte.

The next section of logic performs update operations to the StateStore. This action is controlled by the update state machine. The update sequence starts when the end of a packet is reached and either the upd_required signal or the upd_newflow signal indicate that context information should be saved. Once the sequence is started, it continues to cycle to a new state on each clock cycle until the sequence is completed. This sequence involves sending the flow identifier, followed by four 64-bit words of per-flow context information. The ss_out signals temporarily store the last 32 bytes of the check string. This allows the check string to be loaded with the context information of the next packet without having to wait for the context update operation to complete. The state_store_update process drives the StateStore signals utilized to save the new per-flow context information.

The next section of logic contains the stream comparison logic. A TCAM approach is used to perform the actual comparisons where each comparator contains a value and a mask. The match is performed by performing a check to see if the equation $((check\ string\ XOR\ value)\ NAND\ mask)\ ==\ (all\ ones)$. The first process controls all of the matching operations by indicating whether a match should be performed on this clock cycle. The next process maintains the match1_value and match1_mask values. These values are updated when the set_state and set_id signals are set. The next eight processes perform the actual string 1 matching. The 256-bit match operation is divided into eight segments in order to reduce the fan-in for the match_found signal. Each of these eight processes match a separate 32 bits of the match1 string and produce a separate match_found1 signal. The match process performs four separate matches, each offset by one byte in order to achieve full coverage for the matching operation. There is a small hole in the logic which may cause a false positive match, but it does not adversely affect the overall operation of the circuit and thus has not been addressed. An identical set of processes manage the matching operations for strings 2, 3, and 4.

The next processes drive the outbound signals by copying the internal packet signals. This induces an extra clock cycle delay in the data, but helps to improve the maximum operating frequency by providing an extra clock cycle for these signals to traverse the FPGA.

The next group of processes is responsible for illuminating the LEDs whenever one of the match strings is detected in a TCP data stream. The first process maintains a free-running counter which is used by the subsequent processes to define the duration that a LED should be lit upon the occurrence of a string match. This is required in order to see the LED (illuminating the LED for 1 clock cycle has no visible effect). The led1_count and the led1_ena signals manage the illumination of LED 1. When all of the match1_found signals are set, the current counter value is stored in the led1_count signal and the LED is illuminated by setting the led1_ena signal. The LED remains illuminated until the counter wraps and returns to the value saved in the led1_count signal. The same operations are repeated to manage LED 2, 3, and 4.

The next series of processes manage the modification of the search strings. The sequence of events is controlled by the set state machine which waits for the SET_ENA signal to be asserted. The first word of SET_DATA contains the number of the string to be modified. The next eight clock cycles contain the string value and the final eight clock cycles contain the string mask. The set_count signal counts through the eight clock cycles associated with the string value and mask. The set_value signal holds the new match value and the set_mask holds the new match mask. If an incomplete set sequence is received, the set state machine returns to the idle state and ignores the command.

The query state machine manages the querying of the current match string values and masks. The query request is initiated by the ControlProcessor component driving the QUERY_ENA signal and providing the identifier of the string to query on the QUERY_DATA signals. Upon receiving the QUERY_ENA signal, the current match string value and mask values are saved into the qry_value and qry_mask signals. During the next eight clock cycles, the match value is clocked over the RESPONSE_DATA signals. The match mask is then clocked over the same response signal lines during the subsequent eight clock cycles. The RESPONSE_ENA and RESPONSE_LAST signals provide extra control information which the ControlProcessor uses to frame the query response.

The final process of the source file manages miscellaneous output signals. These include the output LEDs and a set of match signals which are used by the ControlProcessor to collect string match statistics.

5.4.3 ControlProcessor

The ControlProcessor component is described in the `control_proc.vhd` source file and the overall flow of data is shown in Figure 31. The first several processes copy inbound network data into internal signals and load the packets into the frame FIFO. Once a complete packet has been written to the frame FIFO, one bit is written to the control FIFO to indicate that a complete packet is ready for outbound transmission.

The next couple of processes are responsible for maintaining the interval timer which drives the generation of the trigger event. This operation is identical to that of the ControlProcessor module of the PortTracker application. The trigger event initiates the saving of all statistics counters and the transmission of a statistics packet. The next group of processes are responsible for maintaining statistics counters for each of the four string matches which are tracked by this circuit. Upon reception of the trigger event, these counters are saved (implemented in logic discussed later) and reset.

The packet state machine is responsible for processing match string set commands, match string query requests, and managing the transmission of all outbound traffic (including the forwarding of inbound traffic, responses to commands and queries, and generated statistics packets). Upon detection of the trigger event, the state machine sequences through a set of states which is responsible for generating the statistics packet and sending it out via the outbound interface. When a non-empty condition is detected for the control FIFO, a network packet is read from the frame FIFO and processed. If the packet is determined to contain a command or query for the Scan application, then a sequence of states are entered which either process

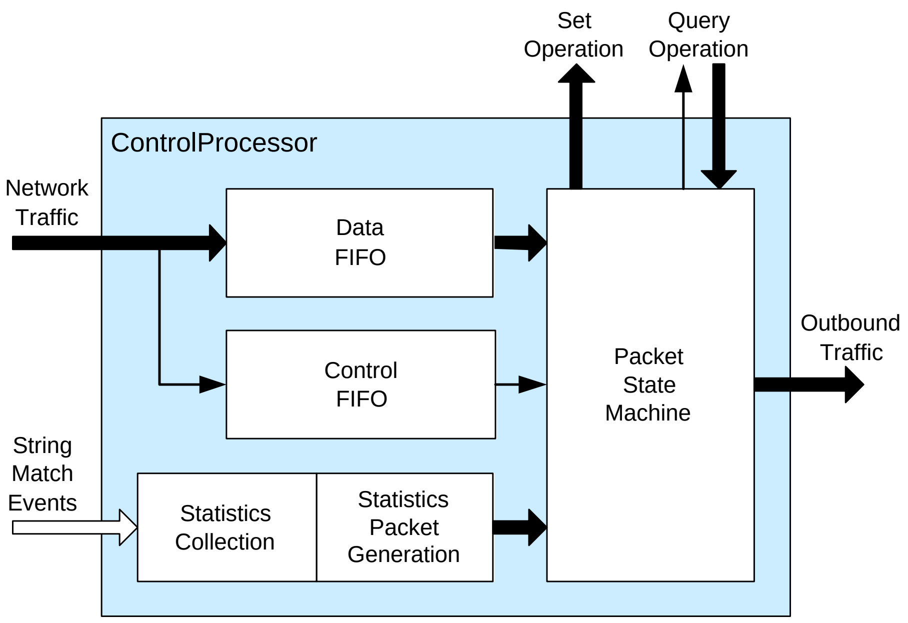


Figure 31: Scan ControlProcessor Layout

the match string set command or the match string query command. This determination is made by looking at the source and destination ports of the incoming packet. If the both of these port numbers are equal to `0x1969`, then the packet state machine will enter the command/query processing states. The state machine also looks at the first word of the UDP payload to see if it contains the *magic* value of `0x23a8d01e`. If this *magic* value is detected, then the next word is assumed to be the command word which determines whether this command is a set or a query. Results of the set or query operation cause the generation of an outbound packet. All other traffic is just transmitted out the output interface. Following the packet state machine are a couple of processes which control the reading of packets from the frame and control FIFOs. The last_read signal marks the end of a packet and is used by the frame FIFO output control process along with the ffifo_ack signal to ensure that all packet data is read from the frame FIFO.

The next process is responsible for taking a snapshot of all the statistic counters and saving the current values when the trigger event fires. This allows the counters to reset and continue counting while the statistics packet is being generated and transmitted.

The format of the statistics packet follows the same format as described in the TCPStats section. The

Scan application generates statistics packet with a stats identifier of two (2). The exact sequence of statistics values transmitted in the statistics packets is listed below:

- cfg1: simulation
- scn1: number of occurrences of match string 1
- scn2: number of occurrences of match string 2
- scn3: number of occurrences of match string 3
- scn4: number of occurrences of match string 4

The output process is responsible for driving the outbound signals with network packets. There are four basic packet types. The first is statistics packets, the second is a response to the set command, the third is a response to the query command, and the fourth is passthrough traffic. The ControlProcessor statistic packet generation logic is identical to the statistic packet generation logic contained in the PortTracker and TCP-Processor circuits, except for the individual statistics values. In order to simplify some of the command processing logic, all inbound packets to the ControlProcessor are assumed to be control packets. For all of these packets, the output process replaces the source IP address with the destination IP address and inserts the specified stats source IP address into the source IP address field of the packet. Additionally, the packet length of the UDP header is changed to 0x54 to correspond to the length of a response/confirmation packet. During the command state, the request/response bit is set in the control word. The result of the query operation is output during the WFR and the Response states. The layout of the Scan control packet format can be seen in Figure 32. All other inbound packet data is forwarded to the output interface in a normal fashion.

Following the output process, there are a couple of processes which aid in the processing of control packets. These processes count the number of IP header words, store the destination IP address so that it can be used to replace the source IP address, and generate statistics packets. The final two processes drive the SET and QUERY interface signals to the ScanApp component. These signals actually control the set and query operations.

5.4.4 StateStore

The StateStore component provides per-flow context storage services for the Scan application. It exposes simple request, response and update interfaces to the application. In addition, all interactions with the

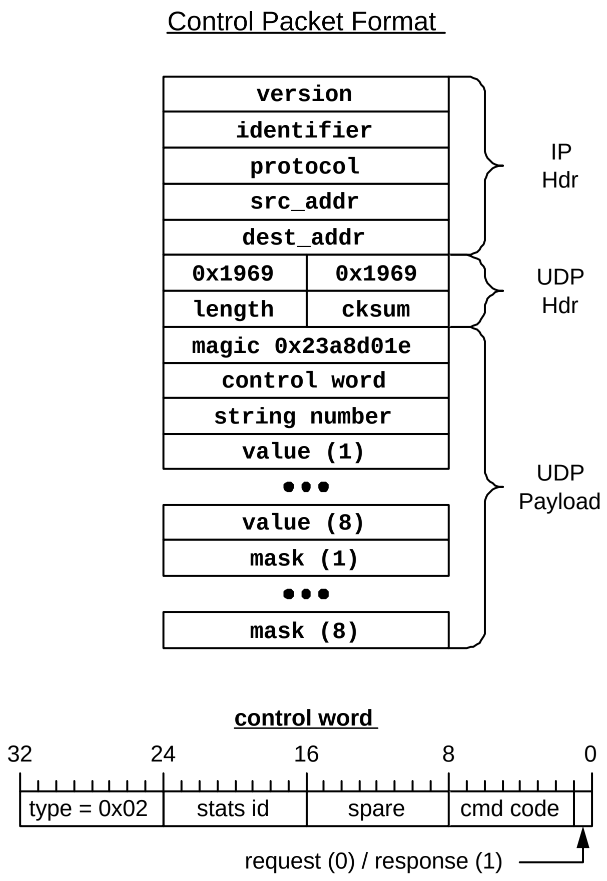


Figure 32: Scan Control Packet Format

SDRAM controller to store and retrieve data are handled by this component. The StateStore is modelled after the StateStoreMgr of the TCP-Processor, but is greatly simplified because it is not required to perform any of the flow classification operations. It is only concerned with providing data storage and retrieval services for the Scan application. In order to simplify the operation of the StateStore component, 64-bit wide interfaces are utilized for both the response and update data.

The layout of the StateStore component is shown in Figure 33. Its three state machines drive all of the operations of the StateStore. The source VHDL code for the StateStore component is in `statestore.vhd`.

The first process in the source file drives the response interface signals. Data returned from SDRAM

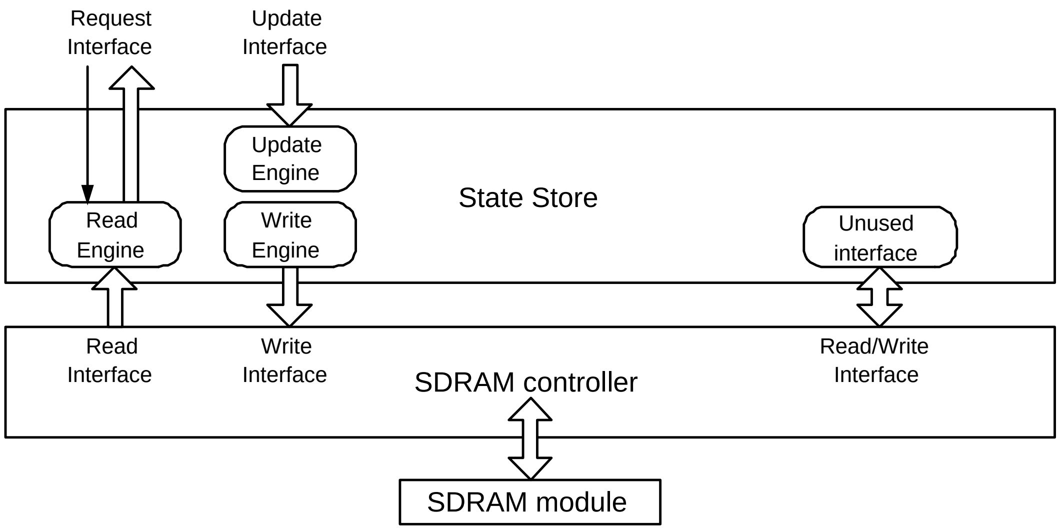


Figure 33: Layout of StateStore

is copied to the response interface signals and passed to the ScanApp component. The next couple of processes implement the read state machine. Upon receiving a request from the ScanApp, the state machine sequences through the states necessary to perform the memory read operation and receive the response. The read engine drives the read control signals of the SDRAM controller. The SS_REQ_DATA signal contains the flow identifier of the context to retrieve. This flow identifier is used as a direct address into memory and the read engine initiates the 32-byte read operation starting at this address. This flow identifier (memory address) is stored in the read_addr signal for the duration of the read operation. Once the presence of the read response has been signaled by the memory controller, the take_count signal counts the data words returned from memory. As this data is being returned from memory, it is copied to the response signals and passed to the ScanApp component.

The update state machine is responsible for processing data associated with an update request. A separate state machine and buffer are required here because write data cannot be written directly to the SDRAM controller without first setting up the write operation. Upon detection of an update command, the new context information is written to the writeback_ram for buffering while the write operation is being setup. The update_count signal keeps track of the number of received from the ScanApp and is also used as the address of where to write the data in the writeback_ram block.

The write state machine is responsible for managing write operations to the SDRAM controller. The initial sequence of states out of reset perform memory initialization tasks by writing zeros to all memory locations. After memory initialization is completed, the write state machine enters the Idle state, waiting for update requests to be received. Upon reception of the SS_UPD_VALID signal, the state machine enters the Req state to initiate the write request to SDRAM. Based on responses from the SDRAM controller, the write state machine traverses through the remaining states which complete the write operation. The reading of the updated per-flow context information from the writeback_ram is controlled by the next process. A separate interface is used so that writes to the writeback_ram can occur at the same time as read operations. The SDRAM controller interface signals are driven by the write_engine. Operations include the initialization (zeroing) of all memory locations and the writing of context information to external memory. The address of the appropriate context information storage location in memory is held in the write_addr signal for the duration of the write operation. Following the write_addr signal, the processes that control the memory initialization operation provide an incrementing memory address and a completion flag. The next process counts the number of words read from the writeback_ram and written to external SDRAM. This count value is used as the read address for the writeback_ram. The final process in the source file drives the ready signal. It indicates when the memory initialization operation has been completed.

6 Usage

This section describes in detail how to use the TCP-Processor circuit. It provides information on how to compile, simulate, synthesize, place & route, and run the various TCP-Processing circuits. This documentation assumes that the user is familiar with ModelSim, Synplicity, Xilinx tools, cygwin, gcc, and a text editor for modifying files. These circuits were develop utilizing ModelSim 5.8SE, Synplicity 7.2Pro, and Xilinx ISE 6.1i.

Figure 34A shows a logical diagram of the encoding and decoding operations as traffic is passed between the StreamExtract, the PortTracker, and the Scan circuits. Figure 34B shows a physical layout of how the FPX devices can be stacked in order to match the logical layout.

The circuits easily achieve a post place & route clock frequency of 55 MHz targeting a Xilinx Virtex XCV2000E-6. A maximum clock frequency of 87 MHz has been achieved for the StreamExtract circuit when targeting the -8 speed grade part.

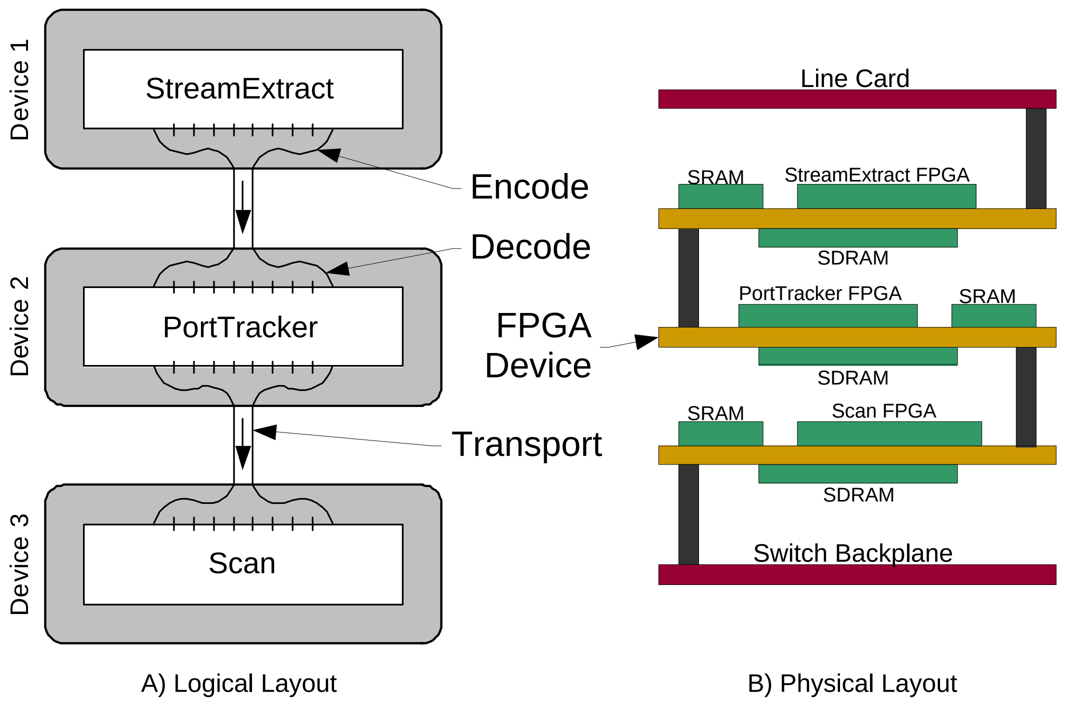


Figure 34: Multidevice Circuit Layout

6.1 StreamExtract

The distribution for the StreamExtract circuit can be found at the URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source /streamextract_source_v2.zip`. Unzip the distribution into an appropriately named directory. Edit the file `vhdl/streamextract_module.vhd` and modify the configuration parameters as required by the target environment. See the previous section on configuration parameters for a full discussion on the various parameters and the effect that they have on the operation of the circuit.

6.1.1 Compile

The compilation of the VHDL source is a very straightforward process. A Makefile compiles the code, assuming the machine has been properly setup with the aforementioned tools, environment variables, licenses, and search paths. The command is shown below:

```
$ make compile
```

This operation should complete successfully. If errors occur, they can be resolved using the information provided in the error message.

6.1.2 Generating Simulation Input Files

Prior to simulation, generate a simulation input file. The easiest method is to use the `tcpdump` [15] command to capture network traffic, such as HTTP web requests. This can be accomplished with a laptop computer connected to a Linux machine which acts as a gateway to the Internet. A `tcpdump` command like the one below will capture the network traffic to a file.

```
$ tcpdump -i eth1 -X -s 1514 host 192.168.200.66 > tmp.dmp
```

Change the `tcpdump` parameters in order to obtain the proper network capture. Specify the `-X` flag to indicate a hexadecimal output format and the `-s 1514` parameter to capture the complete contents of each packet.

Next, convert the `tcpdump` output format into a ModelSim input file. This task can be accomplished using the `dmp2tbp` application. Information on the dmp2tbp utility along with links to the executable and source can be found at:
`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/dmp2tbp.html`.

The output of the `dmp2tbp` program should be a file called INPUT_CELLS.TBP. This file is used as an input to the IPTESTBENCH [16] program. Copy the INPUT_CELLS.TBP file to the `sim` directory and run the command:

```
$ cd sim
$ make input_cell
```

This generates a ModelSim input file called INPUT_CELLS.DAT. With this file, simulations can be performed. Note that the `dmp2tbp` application inserts two header words before the start of IP header so the **NUM_PRE_HDR_WRDS** configuration parameter will need to be set to a value of 2.

6.1.3 Simulate

Once the VHDL source code is compiled, and a simulation input file is generated, it is possible to perform functional simulations. This is accomplished by entering following command:

```
$ make sim
```

This brings up ModelSim, a Signals window, a Waveform window, and runs the circuit for 65 microseconds. If problems occur, check for errors on the command line and in the ModelSim output window. Resolve any problems and rerun the simulation. At this point, use the normal ModelSim functions to interrogate the operation of the circuit. Use the ModelSim `run` command to simulate larger input files.

The StreamExtract testbench generates a file called LC_CELLSOUT.DAT which contains encoded TCP stream data. This file can be used as the simulation input for circuits like the PortTracker and Scan application which use the TCPLiteWrappers.

6.1.4 Synthesize

Prior to synthesis, edit the `vhdl/streamextract_module.vhd` source file and change the configuration parameters to reflect the target environment. Most notably, change the *Simulation* parameter to zero which indicates that all of memory should be initialized and utilized during the operation of the circuit. At this point, either run Synplicity in batch mode or interactive mode. For batch mode operation, enter the following command:

```
$ make syn
```

For interactive mode, start up the Synplify Pro executable, load the `sim/StreamExtract.prj` project file, and click *run*. Regardless of the mode of operation, check the `syn/streamextract /streamextract.srr` log file and look for warnings, errors, and the final operation frequency. Edit the implementation options to target the correct speed grade and clock frequency for the target device.

6.1.5 Place & Route

The place & route operations are executed in batch mode and are initiated by the following make file:

```
$ make build
```

To change the target device, edit the file `syn/rad-xcve2000-64MB/build` and change *part* variable to equal the appropriate device. To change the target clock frequency, edit the file `syn /rad-xcve2000-64MB/fpx.ucf`, go to the bottom of the document and modify the *timespec* and *offset* variables to correspond to the desired clock frequency.

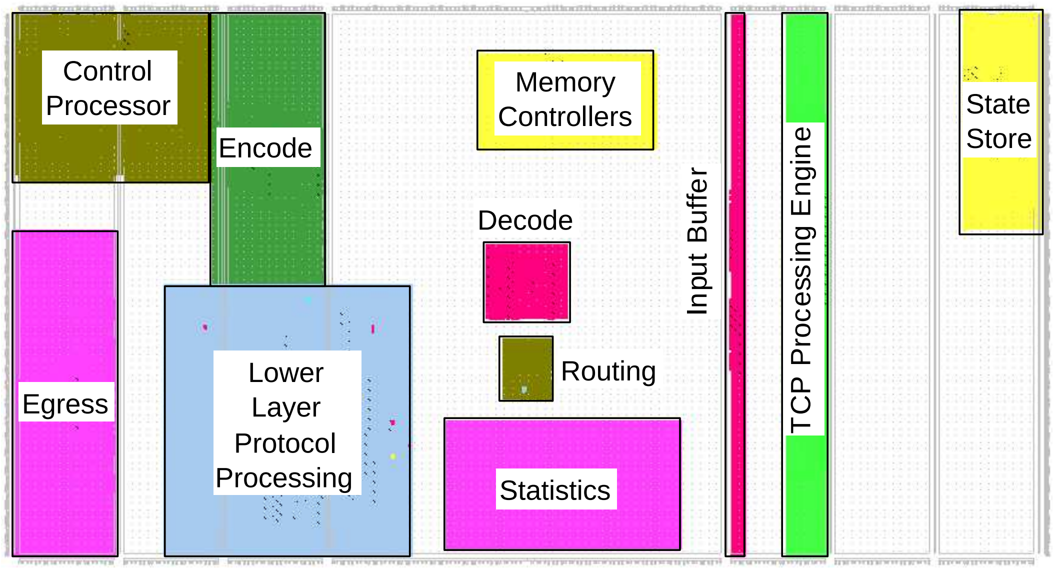


Figure 35: StreamExtract Circuit Layout on Xilinx XCV2000E

The final step in the build process is the generation of the bit file. Prior to using the bit file, verify that the operational clock frequency of the newly created circuit meets or exceeds the design constraints. The timing report for the circuit can be found in the file `syn/rad-xcve2000-64MB/streamextract.twr`. If the circuit achieves the desired performance levels, load the circuit into an FPGA and start processing traffic. The bit file can be found at `syn/rad-xcve2000-64MB/streamextract.bit`.

The StreamExtract circuit has a post place & route frequency of 87.085 MHz when targeting the Xilinx Virtex XCV2000E-8 FPGA. The circuit utilizes 41% (7986/19200) of the slices and 70% (112/160) of the block RAMs. The device is capable of processing 3 million 64-byte packets per second. Figure 35 shows a diagram of the StreamExtract circuit layout on a Xilinx Virtex 2000E FPGA. The layout is divided into regions which roughly correspond to the various components of the circuit.

6.1.6 Setup

Now that the circuit has been built, it is ready to be loaded into an FPX device. To accomplish this task, use the NCHARGE application to perform remote configuration, setup and programming of the FPX device.

Prior to sending traffic through the device, set up the NID routes as shown below. Figure 36 shows

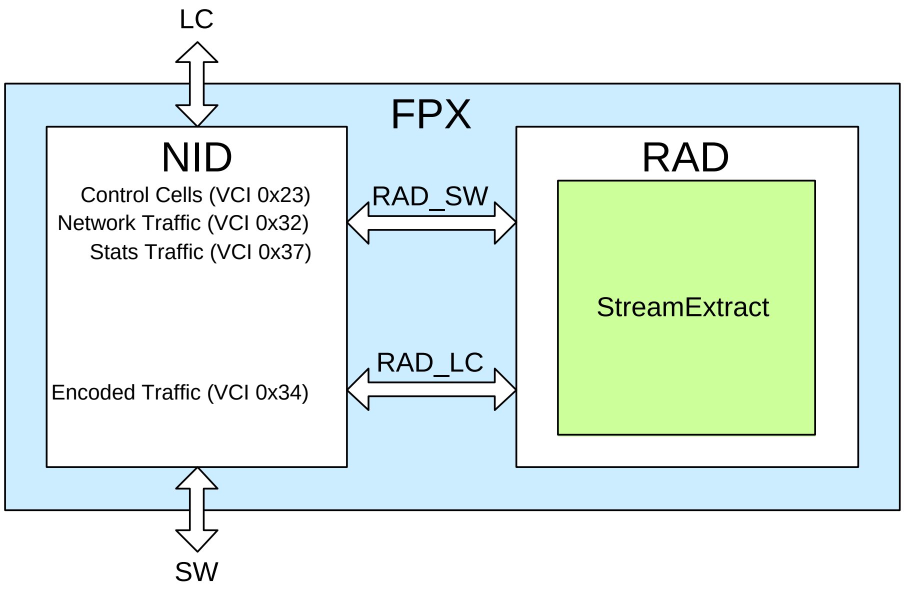


Figure 36: StreamExtract NID Routes

the virtual circuit routes between the NID and the RAD for the StreamExtract circuit. Raw network traffic should be routed through the RAD_SW interface over VPI 0x0 VCI 0x32. In addition, control cell traffic (VPI 0x0 VCI 0x23) and statistic traffic generated by the StreamExtract circuit (VPI 0x0 VCI 0x37) should be routed over the RAD_SW interface. The virtual circuit used by the statistics module can be changed via the previously discussed StreamExtract configuration parameters. Encoded TCP stream data is passed through the RAD_LC interface using VPI 0x0 VCI 0x34.

6.2 Scan & PortTracker

Due to their similarity, the discussions on the Scan and PortTracker applications have been grouped together. The distribution for the Scan circuit can be found at the URL:
`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source/`
`scan_source.zip`
The distribution for the PortTracker circuit can be found at the URL:
`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source`

`/porttracker_source.zip`
Unzip the distributions into appropriately named directories. Edit the `vhdl/scan_module.vhd` and `vhdl/porttracker_module.vhd` source files and modify the configuration parameters as required by the target environment. See the previous section on configuration parameters for a full discussion on the various parameters and the effect that they have on the operation of the circuits.

6.2.1 Compile

The compilation of the VHDL source is a very straightforward process. A Makefile compiles the code. The command is shown below:

```
$ make compile
```

If errors occur, resolve the errors utilizing information provided in the error message.

6.2.2 Generating Simulation Input Files

Prior to simulation, generate a simulation input file which contains encoded TCP stream data. The simulation of the StreamExtract circuit generates the appropriately formatted simulation input file. Copy the `sim/LC_CELLSOUT.DAT` file from the StreamExtract project to the `sim/INPUT_CELLS_LC.DAT` file in the current project. A special input file can be created to test the control and stats features by utilizing the same methods as outlined in the StreamExtract project for creating an INPUT_CELLS.DAT file.

6.2.3 Simulate

After compiling the VHDL source code and generating a simulation input file, functional simulations can be performed. To accomplish this, enter the following command:

```
$ make sim
```

This brings up ModelSim, a Signals window, a Waveform window, and runs the circuit for a short period of time. Longer simulation runs can be produced by using the ModelSim `run` command to simulate more clock cycles.

6.2.4 Synthesize

Prior to synthesis, edit the `vhdl/scan_module.vhd` and `vhdl/porttracker_module.vhd` source files and change the configuration parameters to reflect the target environment. Most notably, change the *Simulation* parameter to zero which indicates that all of memory should be initialized and utilized during the operation of the circuit. At this point, either run Synplicity in batch mode or interactive mode. For batch mode operation, enter the following command:

```
$ make syn
```

For interactive mode, start up the Synplify Pro executable, load the `sim/Scan.prj` or `sim/PortTracker.prj` project file, and click *run*. Regardless of the mode of operation, check the `syn/scan/scan.srr` or `syn/porttracker/porttracker.srr` log files and look for warnings, errors, and the final operation frequency. Edit the implementation options to target the correct speed grade and clock frequency for the target device.

6.2.5 Place & Route

The place & route operations are executed in batch mode and are initiated by the following command:

```
$ make build
```

To change the target device, edit the file `syn/rad-xcve2000-64MB/build` and change *part* variable to equal the appropriate device. To change the target clock frequency, edit the file `syn/rad-xcve2000-64MB/fpx.ucf`, go to the bottom of the document and modify the *timespec* and *offset* variables to correspond to the desired clock frequency.

The final set in the build process is the generation of the bit file. Prior to using the bit file, verify that the operational clock frequency of the newly created circuit meets or exceeds the design constraints. The timing report for the circuit can be found in the file `syn/rad-xcve2000-64MB/scan.twr` or `syn/rad-xcve2000-64MB/porttracker.twr`. If the circuit achieved the desired performance levels, load the circuit into a FPGA and start processing traffic. The bit file can be found at `syn/rad-xcve2000-64MB/scan.bit` or `syn/rad-xcve2000-64MB/porttracker.bit`, depending on which application is being built.

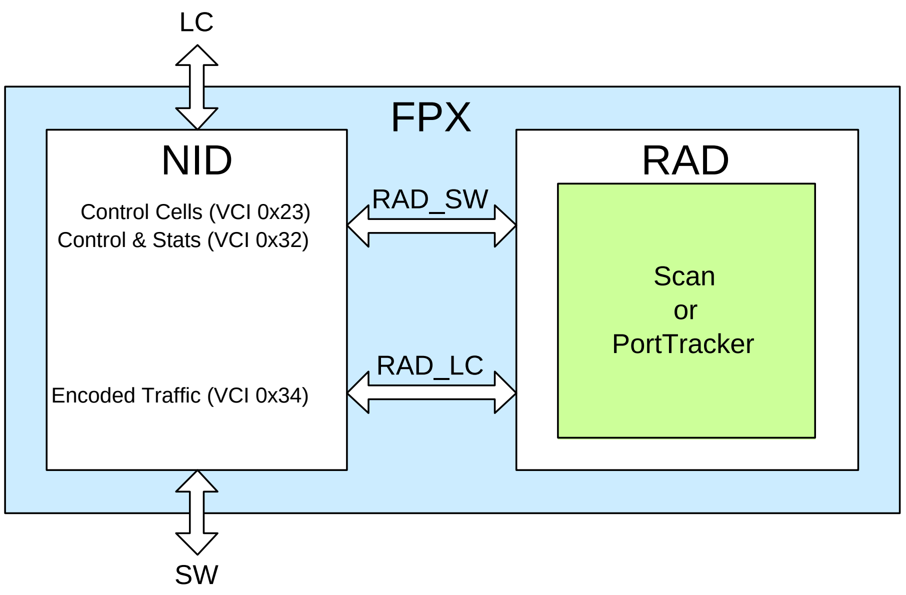


Figure 37: Scan/PortTracker NID Routes

6.2.6 Setup

Now that the circuit has been built, it is ready to be loaded into an FPX device. The steps required to perform this task are identical to those of the StreamExtract circuit. Next, set up the NID routes. The NID routes for both the Scan and the PortTracker are the same. Figure 37 shows the virtual circuit routes between the NID and the RAD for the Scan and PortTracker circuits. Network traffic carrying control and statistics information should be routed through the RAD_SW interface over VPI 0x0 VCI 0x32. Control cell traffic should be routed to the RAD_SW interface over VPI 0x0 VCI 0x23. Encoded TCP stream data is passed through the RAD_LC interface using VPI 0x0 VCI 0x34.

6.3 New Applications

Prior to starting development of a new TCP stream data processing application, it is important to analyze the environment, resources, and size of the target application. The StreamExtract circuit, the PortTracker circuit, and the Scan circuit are three recommended starting points for new application development. All of these projects are structured in a similar manner so that the transitions from one environment to another are easily achieved without requiring mastery of different interfaces, file structures, or build processes.

The StreamExtract circuit should be utilized if a small footprint solution is desired. This allows the TCP stream processing application to coexist in the same circuit as the TCP-Processor, thus utilizing only one device. The PortTracker circuit is a good example of a multi-device solution where the TCP-Processor operates on one device and the TCP stream processing application operates on one or more other devices. Since a separate device is utilized for the stream processing application, more FPGA resources are available to perform processing. The Scan circuit is similar to the PortTracker circuit, except it also contains additional logic to store and retrieve per-flow context information in off-chip SDRAM devices. The Scan circuit also contains logic for processing external commands formatted as UDP data packets.

Once the appropriate starting point for the intended TCP stream processing application has been determined, the selected example circuit can be downloaded and extracted in order to begin the desired modifications. If using the StreamExtract circuit as the starting point, edit the file `streamextract.vhd`. Remove the *TCPSerializeEncode* and *TCPSerializeDecode* components and replace them with components specific to the new application. If the Scan or PortTracker circuits are selected as the starting point, edit the `_module.vhd` source file and replace either the ScanApp or PortTrackerApp circuits with logic for the new application. Changes will also be required to the `controlproc.vhd` source file to support different control and statistics features.

6.3.1 Client Interface

The client interface is identical for a TCP stream processing application based on either the StreamExtract, PortTracker, or Scan circuits. Detailed information on this interface was previously described in the section covering the TCPRouting component of the TCP-Processor. This information covers the interface signals, a sample wave form, and details on the data passed through the interface. Another excellent approach to gaining familiarity with the client interface of the TCP-Processor (and TCPLiteWrappers) is to run simulations with various data sets and inspect the various signal transitions.

6.4 Runtime Setup

The FPX cards on which the TCP processing circuits operate can either be inserted into the WUGS-20 switch environment or the FPX-in-a-Box environment. Runtime setup and configuration information is presented for both of these environments. While the circuits and the FPX cards are identical for both of

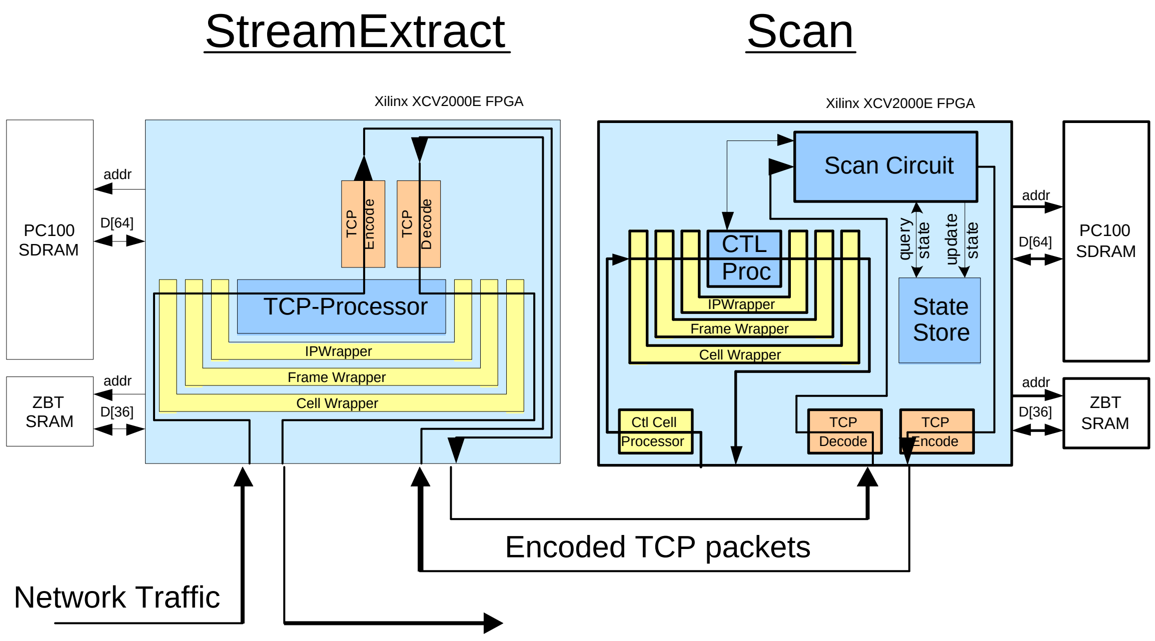


Figure 38: Multidevice Circuit Layout

these environments, there are configuration differences due to the WUGS-20 backplane's ability to route traffic from any port to any other port. The FPX-in-a-Box solution, conversely, relies on the NID to perform all traffic routing functions. The interaction between two FPX devices working together to provide a TCP stream monitoring service is shown in Figure 38. Network traffic processed by the TCP-Processor is encoded and passed to a separate device for additional processing. This second device processes the encoded TCP stream data and searches for digital signatures. The encoded data is passed back to the TCP-Processor for egress processing and to forward data on to the end system or next hop router. This final step is not required when performing passive monitoring because the network packets do not need to be delivered to the end system via this network path.

Figure 39 shows a WUGS-20 switch configured to perform active monitoring of inbound Internet traffic. A laptop computer initiates communications with hosts on the Internet. These communications can include remote logins, file transfers and web surfing. Traffic initiated on the laptop is passed through a network hub and routed to an OC-3 line card installed at Port 1 of the switch. This traffic is passed directly to the OC-3

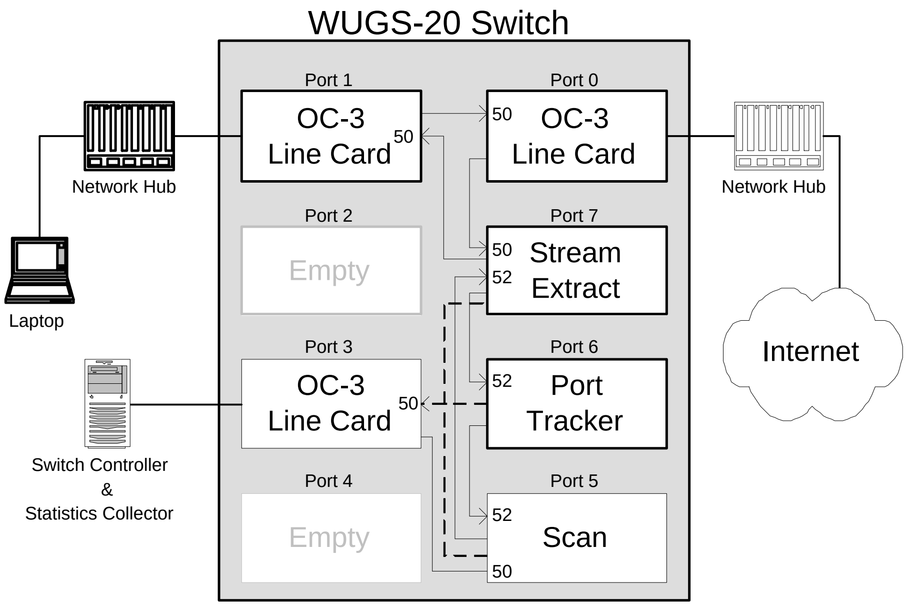


Figure 39: Active Monitoring Switch Configuration

line card at Port 0, transferred to a different network hub and routed to the Internet. Return traffic from the Internet is passed to the line card at Port 0. This network traffic is routed to an FPX card at Port 7 which is running the StreamExtract circuit. Encapsulated TCP stream data is passed to the PortTracker circuit at Port 6 and then to the Scan circuit at Port 5. After the Scan circuit, this encoded network traffic is passed back to the StreamExtract circuit at port 7 which reproduces the original network data packets and passes them to the line card at Port 1 for delivery to the laptop. Statistics information generated by the StreamExtract, the PortTracker and the Scan circuits is passed to the OC-3 Line card at Port 3 which forwards the statistics to a host machine which collects them. The statistics traffic is shown as a dashed line in the figure.

The VCI routes which need to be programmed into the WUGS-20 switch are shown in Table 4. The table is broken into four sections showing routes used for network traffic, encoded network traffic, statistics traffic, and control traffic. The Scan circuit is the only circuit which supports remote management features and therefore only requires control traffic. NID routing information can be found in the previous sections dealing with each of the specific circuits.

Figure 40 shows a WUGS-20 switch configured to perform passive monitoring of network traffic. Mir-

Network Traffic
`Port:  1 VCI: 50 => Port:  0 VCI: 50`
`Port:  0 VCI: 50 => Port:  7 VCI: 50`
`Port:  7 VCI: 50 => Port:  1 VCI: 50`
Encoded Network Traffic
`Port:  7 VCI: 52 => Port:  6 VCI: 52`
`Port:  6 VCI: 52 => Port:  5 VCI: 52`
`Port:  5 VCI: 52 => Port:  7 VCI: 52`
Statistics Traffic
`Port:  7 VCI: 55 => Port:  3 VCI: 50`
`Port:  6 VCI: 50 => Port:  3 VCI: 50`
`Port:  5 VCI: 50 => Port:  3 VCI: 50`
Control Traffic
`Port:  3 VCI: 50 => Port:  5 VCI: 50`

Table 4: Unidirectional Routing Assignments for Active Monitoring Switch Configuration

roring, replication or optical splitting techniques can be used to acquire copies of network traffic for passive monitoring. In this configuration, the traffic to be monitored is passed into the GigE line card at Port 1. Traffic is then passed to the StreamExtract circuit at Port 2 where TCP processing is performed. Encoded network traffic is passed via VCI 52 to the PortTracker circuit at Port 4 and the Scan circuit at Port 6. Since this is a passive monitoring configuration, there is no need to route encoded traffic back to the StreamExtract circuit for egress processing. Statistics information generated by the StreamExtract, the PortTracker and the Scan circuits is passed to the GigE Line card at Port 0 which forwards the statistics to the stats collection machine.

The VCI routes which need to be programmed into the WUGS-20 switch for the passive monitoring configuration are shown in Table 5. The table is broken into four sections showing routes used for network traffic, encoded network traffic, statistics traffic, and control traffic. The Scan circuit is the only circuit which supports remote management features and therefore only needs control traffic. NID routing information can be found in the previous sections dealing with the each of the specific circuits.

The runtime setup for TCP stream processing circuits operating within a FPX-in-a-Box environment is different from that of a WUGS-20 switch environment. The main reason for this is that the backplane for the FPX-in-a-Box solution only provides an electrical signal interconnect between the two stacks of FPX

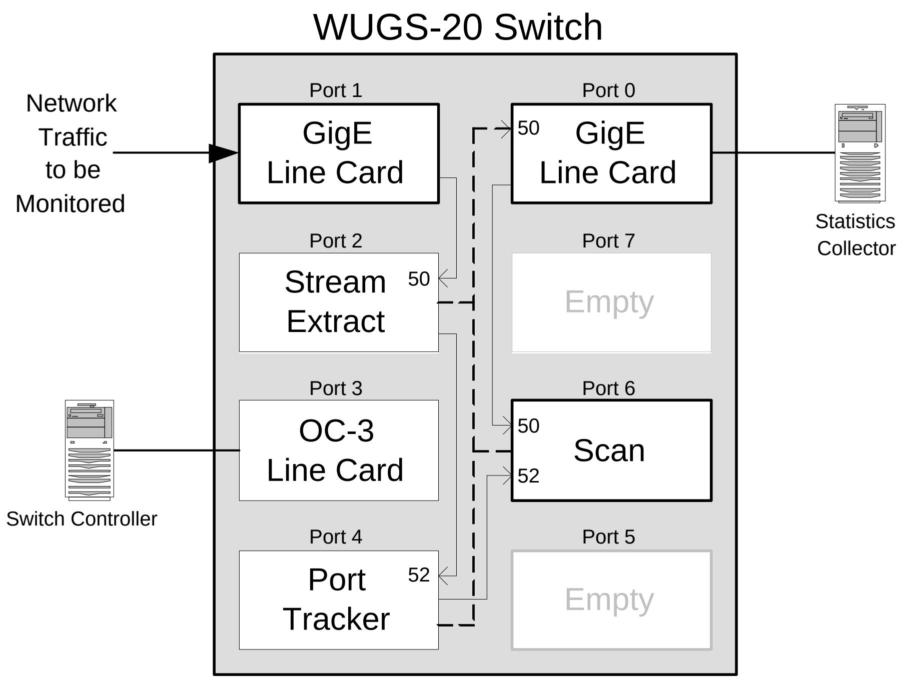


Figure 40: Passive Monitoring Switch Configuration

devices and no routing services. This means that all routing functions must be performed by the NID circuit on each of the FPX devices.

Figure 41 shows a configuration for performing passive monitoring utilizing a FPX-in-a-Box environment. Similar to the passive monitoring configuration in a switch, an external device is required to provide duplicated network traffic to the input line card (located at the top of the stack on the left). Network traffic to be monitored is passed to the StreamExtract circuit for TCP processing. Encoded network packets are passed down to the PortTracker circuit and finally down to the Scan circuit. The NID cannot be configured to act as a data sink for network traffic. For this reason, the Scan circuit will have to be modified in this configuration to not pass encoded network traffic out the RAD_LC interface. Otherwise, an issue develops where the monitored network traffic is reflected inside the FPX-in-a-Box configuration, potentially causing operation problems with other circuits and depleting available bandwidth.

The NID routing information for this configuration is also shown in Figure 41. For each of the FPX

Network Traffic
`Port:  1 VCI: 51 => Port:  2 VCI: 50`
Encoded Network Traffic
`Port:  2 VCI: 52 => Port:  4 VCI: 52`
`Port:  4 VCI: 52 => Port:  6 VCI: 52`
Statistics Traffic
`Port:  2 VCI: 55 => Port:  0 VCI: 50`
`Port:  4 VCI: 50 => Port:  0 VCI: 50`
`Port:  6 VCI: 50 => Port:  0 VCI: 50`
Control Traffic
`Port:  0 VCI: 50 => Port:  6 VCI: 50`

Table 5: Unidirectional Routing Assignments for Passive Monitoring Switch Configuration

cards, the NID can route data from each of the SW, LC, RAD_SW, and RAD_LC input interfaces to any of the same output ports. On FPX 0, the NID should be configured to route VCI 51 (0x33) traffic from the LC interface to the RAD_SW interface. Additionally, VCI 50 (0x32) from the RAD_SW interface and VCI 52 (0x34) from the RAD_LC interface should both be routed to the SW interface. Routes should be setup for the FPX 1 and FPX 2 devices as indicated in the diagram.

6.5 Known Problems

Below is a list of known problems with the current distribution. Most of these problems do not reflect errors in the design or the code, but are due to changes in the tools provided by Xilinx and Synplicity which are incompatible with previous versions of these same tools.

6.5.1 Xilinx ISE 6.2i

Xilinx changed the dual-port RAM block produced by CoreGen. The previous dual-port block RAM memory device contained a version 1 implementation. The current dual-port RAM memory device uses version 5 constructs. The `vhdl/wrappers/IPProcessor/vhdl/outbound.vhd` source file directly references the version 1 component which is no longer supported by newer versions of the tools. Build an identically sized version 5 component and add the vhd and edn files to the project. Xilinx also changed the signal names with the new component. Edit the `outbound.vhd` file and fix up the ram512x16 component

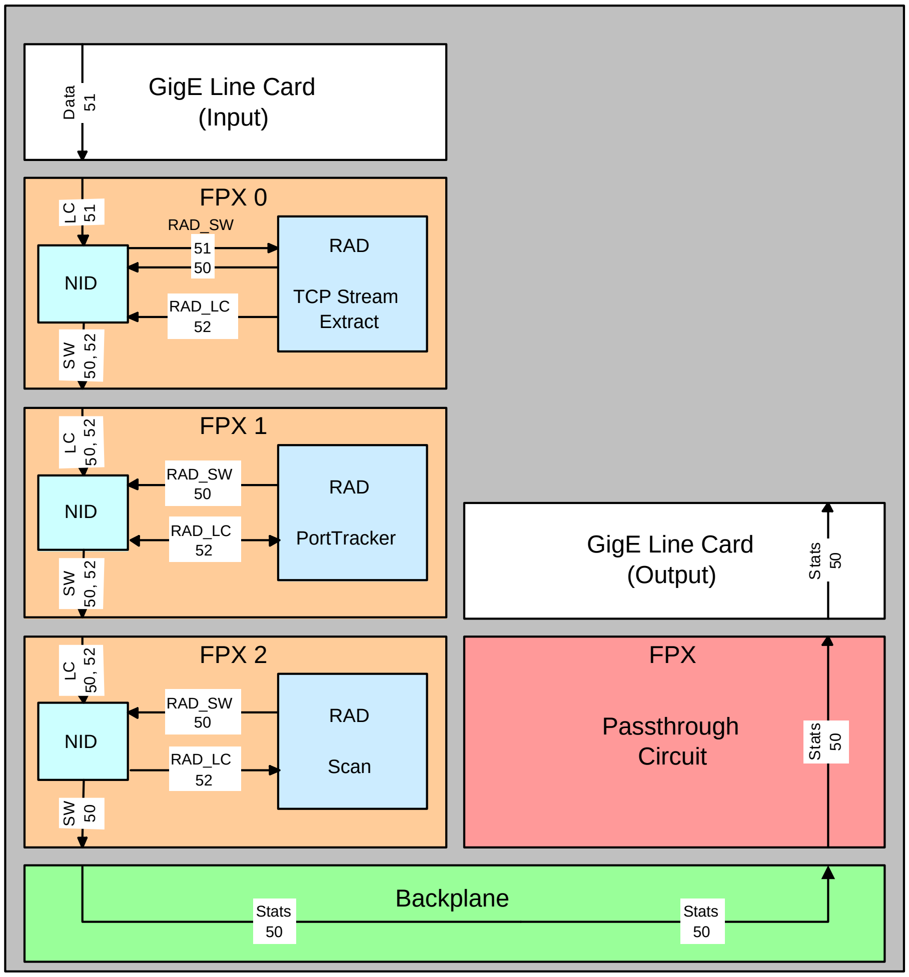


Figure 41: Passive Monitoring FPX-in-a-Box Configuration

and its implementations to correspond to the version 5 CoreGen component.

6.5.2 Synplicity 7.5

Synplicity v7.5 is not compatible with the alias construct used in the `vhdl/wrappers/FrameProcessor/vhdl/aal5dataen.vhd` source file. Previous versions of Synplicity had no problem with this VHDL construct. There are two known workarounds. The first is to switch to an older or newer version of Synplicity (version 7.3 or 7.6 for example). The second workaround is to modify the source file to remove the alias construct and directly address the proper signal names.

6.5.3 Outbound IPWrapper Lockup

There have been several occurrences of a situation where the outbound lower layered protocol wrappers stop processing traffic and de-assert the TCA flow control signal to the TCP-Processor circuit. This problem has been observed when monitoring live network traffic with a configuration similar to that shown in Figure 40. The only difference is that encoded traffic exiting the Scan circuit is routed back to the StreamExtract circuit for egress processing. After several days of processing traffic, the outbound IP Wrapper will de-assert the flow control signal and lock up the whole circuit. It is not clear whether or not this lockup is due to invalid data being sent from the TCP-Processor egress component to the IP Wrapper or a bug in the outbound lower layered protocol wrapper code. The exact sequence of events which causes this problem has not been determined. One plausible explanation for this problem is the presence of an IP packet fragment.

7 Generating Simulation Input Files

The ability to quickly and easily generate simulation input files is paramount to any debugging effort. This section focuses on capturing network traffic and converting it into a simulation input file used by ModelSim and the testbench logic. Simulation input files can be generated by hand, but this is tedious work. The following subsections contain usage information for the `tcpdump`, `dmp2tbp`, IPTESTBENCH, and `sramdump` applications as well as the CellCapture circuit for generating simulation input files.

7.1 tcpdump

The `tcpdump` [15] utility captures and displays the contents of network packets arriving on a network interface on a host machine. A `tcpdump` command like the one below will capture the network traffic to a file.

```
$ tcpdump -i eth1 -X -s 1514 host 192.168.204.100 > tmp.dmp
```

Change the `tcpdump` parameters as needed to specify the proper host. Specify the `-X` flag to indicate a hexadecimal output format and the `-s 1514` parameter to capture the complete contents of each packet. Below is sample of network traffic captured with the aforementioned command:

```
16:04:34.151188 192.168.204.100.3474 >
hacienda.pacific.net.au.http: S 1180902188:1180902188(0) win 16384
<mss 1460,nop, nop,sackOK> (DF)

0x0000 4500 0030 e0a1 4000 7d06 52c9 c0a8 cc64 E..0..@.}.R....d
0x0010 3d08 0048 0d92 0050 4663 232c 0000 0000 =..H...PFc#,....
0x0020 7002 4000 0151 0000 0204 05b4 0101 0402 p.@..Q..........

16:04:34.362576 hacienda.pacific.net.au.http >
192.168.204.100.3474: S 545139513:545139513(0) ack 1180902189 win
5840 <mss 1460,nop,nop,sackOK> (DF)

0x0000 4500 0030 0000 4000 2b06 856b 3d08 0048 E..0..@.+..k=..H
0x0010 c0a8 cc64 0050 0d92 207e 2b39 4663 232d ...d.P...~+9Fc#-
0x0020 7012 16d0 deb8 0000 0204 05b4 0101 0402 p...............

16:04:34.364557 192.168.204.100.3474 >
hacienda.pacific.net.au.http: . ack 1 win 17520 (DF)

0x0000 4500 0028 e0a3 4000 7e06 51cf c0a8 cc64 E..(..@.~.Q....d
0x0010 3d08 0048 0d92 0050 4663 232d 207e 2b3a =..H...PFc#-.~+:
0x0020 5010 4470 dddc 0000 2020 2020 2020      P.Dp..........
```

7.2 dmp2tbp

The `dmp2tbp` application has been developed to convert the `tcpdump` output format into a simulation input file for use by ModelSim. A `dmp2tbp` command takes two parameters, an input file which was generated by a `tcpdump` command and a target output file to contain the IPTESTBENCH input data. The syntax of the application is shown below:

```
    Usage:  dmp2tbp <input dmp file> <output tbp file>
```

Links to the executable and source for the `dmp2tbp` application can be found at:
`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/dmp2tbp.html`.

The converted output of the previous `tcpdump` sample trace is shown below. For each IP packet, a two-word LLC header was prepended to the packet to match the packet format of the switch environment. In order to account for these two header words prior the start of IP header, the **NUM_PRE_HDR_WRDS** configuration parameter will need to be set to a value of 2. The LLC header information can easily be removed by editing the `dmp2tbp` source file.

```
!FRAME 50
AAAA0300
00000800
45000030
e0a14000
7d0652c9
c0a8cc64
3d080048
0d920050
4663232c
00000000
70024000
01510000
020405b4
01010402
!FRAME 50
AAAA0300
00000800
45000030
00004000
2b06856b
3d080048
c0a8cc64
00500d92
207e2b39
4663232d
701216d0
deb80000
020405b4
01010402
!FRAME 50
AAAA0300
00000800
45000028
e0a34000
7e0651cf
c0a8cc64
3d080048
0d920050
4663232d
207e2b3a
50104470
dddc0000
20202020
20200000
```

7.3 IPTESTBENCH

The `ip2fake` executable of the IPTESTBENCH [16] utility converts the output of the `dmp2tbp` application into the simulation input file used by ModelSim. Rename the file generated by the `dmp2tbp` program to INPUT_CELLS.TBP and copy the file to the `sim` directory. Run the following commands to convert the .TBP file to the ModelSim .DAT input file:

```
$ cd sim
$ make input_cell
```

A file called INPUT_CELLS.DAT is generated which contains properly formatted packets which can be used to perform simulations. The source code and make files for IPTESTBENCH are included with each circuit design in the `iptestbench` subdirectory.

7.4 sramdump

The `sramdump` utility reads data out of SRAM and writes the results to a disk file. To accomplish this task, the `sramdump` application communicates with NCHARGE over a TCP socket connection. NCHARGE in turn communicates with the Control Cell Processor (CCP) circuit programmed into a FPX card. This communication utilizes ATM control cells to read the SRAM memory of the FPX card. Results are passed to NCHARGE and then the `sramdump` application via the return path. Figure 42 shows a diagram of this communication path. The `sramdump` utility requires that the CCP circuit be built into the RAD application and that the CCP control cells (typically VCI 0x23) be correctly routed through the switch, the NID and the RAD. Links to the executable and source for the `sramdump` application can be found at:
`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/sramdump.html`.

The `sramdump` program automatically detects the endianess of the host machine and formats output so that the resulting dump file is consistent, regardless which machine on which the utility was run . There are various dumping options, a binary dump [useful as input to other analysis programs], an ASCII dump [good for text data], a dump format [shows both hexidecimal and ASCII representations of data], and an option to generate IPTESTBENCH input files.

SRAM memory on the FPX platform is 36 bits wide. Depending on the applicaitons's use of this memory, data can either be read out in 32-bit segments (ignoring the upper 4 bits) or in 36-bit segments. When reading 36-bit data, the high nibble is normally stored as a byte-sized value. The ”-e” option will

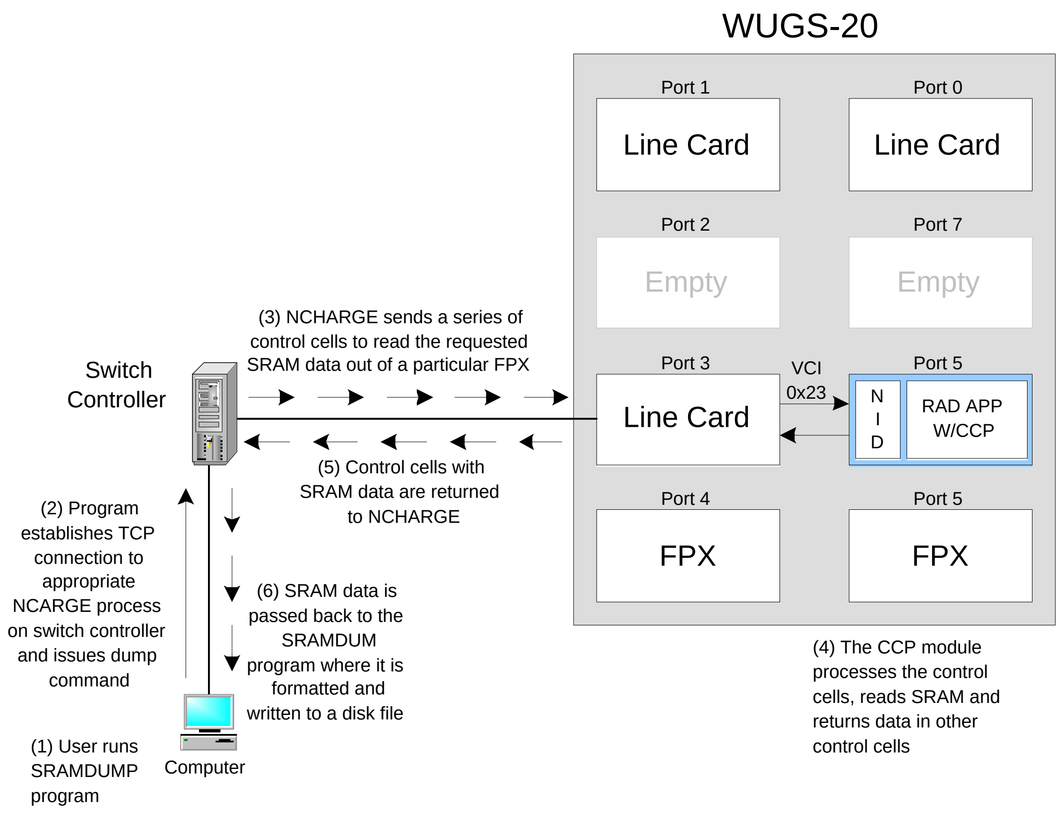

WUGS-20
Port 1
Line Card
Port 0
Line Card
Port 2
Empty
Port 7
Empty
Port 3
Line Card
VCI
0x23
Port 5
N
I
D
RAD APP
W/CCP
Port 4
FPX
Port 5
FPX
Switch
Controller
(3) NCHARGE sends a series of control cells to read the requested SRAM data out of a particular FPX
(5) Control cells with SRAM data are returned to NCHARGE
(2) Program establishes TCP connection to appropriate NCARGE process on switch controller and issues dump command
(6) SRAM data is passed back to the SRAMDUM program where it is formatted and written to a disk file
(4) The CCP module processes the control cells, reads SRAM and returns data in other control cells
(1) User runs SRAMDUMP program
Computer

Figure 42: Operation of SRAMDUMP Utility

expand the output and represent this as a 4-byte quantity such that the low bit of each byte will represent the value of one of four high bits. This formatting option makes it easier to interpret a series of 36-bit data values.

In order to generate data formatted as AAL5 frames for input into IPTESTBENCH, data must be written in a special SRAM frame format. Likewise, in order to generate data formatted as ATM cells for input into ModelSim, data must be written in a special SRAM cell format. Figure 43 shows the specific SRAM frame and cell formats. Output files generated using the **-t F** or **-t C** options can be used as the input file to IPTESTBENCH for creating simulation input files for ModelSim.

Usage information for the `sramdump` application is shown below:

```
Usage: sramdump [OPTIONS]
```

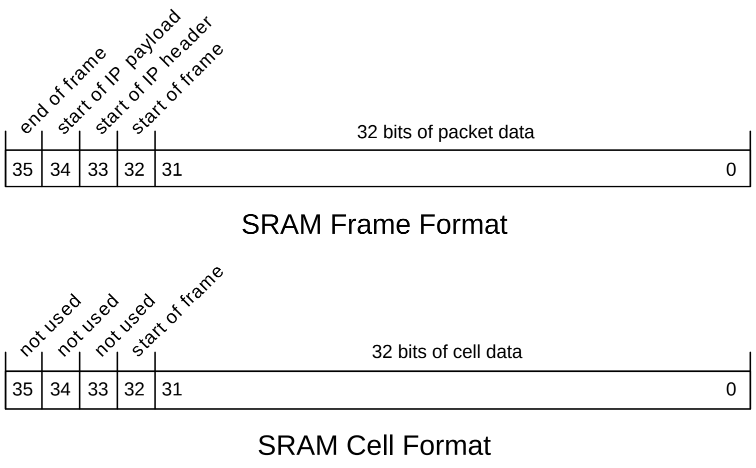
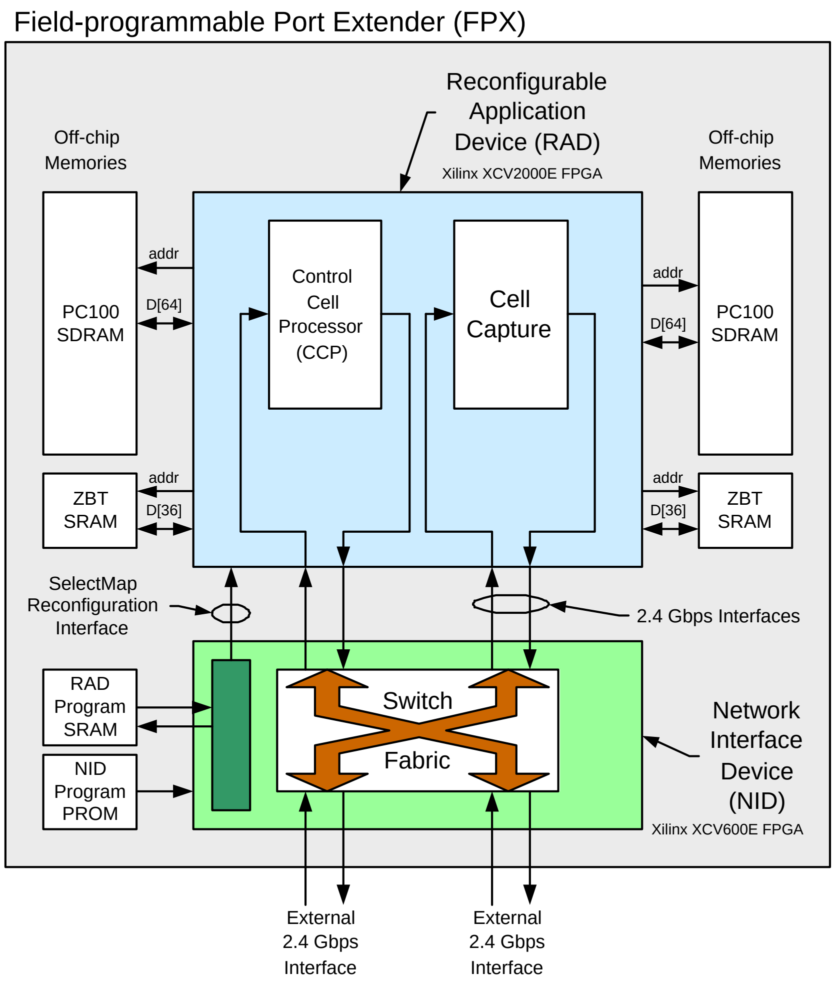


Figure 43: SRAM data Formats

```
-h <string> hostname of switch controller: [illiac.arl.wustl.edu]
-c <num> card number (0 - 7): [0]
-b <num> memory bank (0 - 1): [0]
-a <num> address to start reading (in hex): [0]
-n <num> number of words to read: [1]
-o <string> output file: [sram.dat]
-t <char> output type (B-binary, A-ascii, D-dump, F-frames, C-cells): [B]
-f <char> dump format (B-big endian, L-little endian): [B]
-w <num> dump column width: 4
-s <num> dump separations (spaces every n bytes): [4]
-l <num> length of word (32, 36): [32]
-e expanded 36 bit format (high 4 bits expanded to low bit of 4 byte field
-? this screen
```

7.5 Cell Capture

The CellCapture circuit provides an easy mechanism for capturing traffic with an FPX card. The circuit writes captured ATM cells to the SRAM0 memory device and then to the SRAM1 memory device. Each SRAM unit contains 256 thousand 36-bit memory locations which can store 18,724.5 cells. The CellCapture circuit will store the first 37,449 cells that are passed into the RAD_LC interface. Because the SRAM devices support memory operations on every clock cycle, the circuit is capable of capturing traffic at full line rates (writing data to memory on every clock cycle). The circuit synthesizes at 100MHz which supports the fastest



Field-programmable Port Extender (FPX)
Reconfigurable Application Device (RAD)
Xilinx XCV2000E FPGA
Off-chip Memories
Off-chip Memories
PC100 SDRAM
addr
D[64]
Control Cell Processor (CCP)
Cell Capture
addr
D[64]
PC100 SDRAM
ZBT SRAM
addr
D[36]
addr
D[36]
ZBT SRAM
SelectMap Reconfiguration Interface
2.4 Gbps Interfaces
RAD Program SRAM
NID Program PROM
Switch
Fabric
Network Interface Device (NID)
Xilinx XCV600E FPGA
External 2.4 Gbps Interface
External 2.4 Gbps Interface

Figure 44: CellCapture Circuit Layout

traffic rates possible on the FPX platform. Figure 44 shows a layout of the CellCapture circuit.

The CellCapture circuit is structured in a similar manner to all other circuits described in this document. Links to the project source for the CellCapture circuit can be found at: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp /cell_capture.html`. The steps required to build an executable *bit* file from the source are listed below:

```
$ make compile
$ make syn
$ make build
```

To utilize this circuit, route control cell traffic (VCI 0x23) into the RAD_SW port and route traffic to be captured (the VCI(s) of interest) into the RAD_LC port. The RAD-controlled LEDs 1 & 2 blink continuously to indicate that the circuit is operating. The RAD-controlled LEDs 3 & 4 illuminate when SRAM bank 0 and 1 respectively, are full of data. LED 3 will always illuminate before LED 4 because the circuit fills up SRAM bank 0 before writing to SRAM bank 1.

After capturing network traffic, the `sramdump` utility can be used to retrieve the captured data. The resulting data file can be converted into a ModelSim input file utilizing IPTESTBENCH. Attempts to read data from a SRAM before it is full of data will cause the CCP read operations to SRAM take priority over the CellCapture writes. This implies that inconsistent network captures may be produced if a SRAM read operation occurs at the same time as a SRAM write operation. This circuit is designed to work with the `sramdump` utility. Use the -l 36 -t C -e options with the `sramdump` command to generate ModelSim input files.

7.6 Internal Data Captures

The internal data captures supported by many of the TCP-Processor circuits provide an excellent method for gaining insight into the internal operation of the circuits. These data captures can be extracted from the FPX devices utilizing the `sramdump` routine and converted into simulation input files utilizing IPTESTBENCH. The TCPInbuf, TCPEgress, TCPSerializeEncode, TCPSerializeDecode, TCPDeserialize, and TCPReserialize components all support the storing of network traffic to SRAM memory devices. The TCPInbuf and the TCPEgress components utilize the SRAM frame format for storing network traffic. The TCPSerializeEncode, TCPSerializeDecode, TCPDeserialize, and TCPReserialize components utilize the SRAM cell format for storing network traffic.

8 Conclusion

This document provides a comprehensive look at the TCP-Processor technology and related hardware circuits. Intricate details pertaining to the design, implementation and operation of the TCP-Processor circuit are covered. Additional information is provided that describes the data encoding and decoding process used to support TCP stream processing across multiple devices. Two example applications are described which process TCP stream content from the TCP-Processor. Usage information is also provided which explains how to compile, simulate, sythesize, and place & route the circuit designs. Hardware setup and configuration information is also provided for each of the circuits. A special section covers the various methods and associated tools available to create simulation input files. The appendix describes the utilities available for charting statistical information collected by the various TCP stream processing circuits.

Additional information regarding the TCP-Processor and related technologies can be found at the following web sites:

```
http://www.arl.wustl.edu/˜dvs1
http://www.arl.wustl.edu/projects/fpx/reconfig.htm
http://www.arl.wustl.edu/projects/fpx/projects/tcp/index.html
http://www.arl.wustl.edu/projects/fpx/fpx\_internal/tcp
```

9 Appendix

This appendix contains information on software applications which collect, chart, and redistribute statistics information generated by the various circuits mentioned in this document. These applications directly process the statistics packets generated by the hardware circuits. The statistics packets are transmitted as UDP datagrams, with the packet payload format shown in Table 6.

31 24	23 16	15 8	7 0
Statistics Type	Statistics Identifier	Packet Number	Number of Entries
32-bit statistics value 1			
32-bit statistics value 2			
32-bit statistics value 3			
…			

Table 6: Statistics Format

9.1 StatsCollector

The StatsCollector is a C program which gathers statistics information and spools the data to disk files. A new disk file is created every night so that the data sets can be easily managed. The StatsCollector can also be configured to rebroadcast statistics packets to a maximum of five separate destinations. This aids in the dissemination of statistical information to a wide audience. Figure 45 shows an operational summary of the StatsCollector.

The source for the StatsCollector application can be found at the following web site: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp /statscollector.html`. Usage information for the StatsCollector application is shown below:

```
Usage: statscollector [OPTION]
         -p <num>           UDP port on which to listen for statistics information
         -r <string>:<num>  remote IP host & UDP port on which to re-transmit stats
                            (up to 5 remote hosts can be specified)
         -s                 silent - do not capture statistics to disk file
         -?                 this screen
```

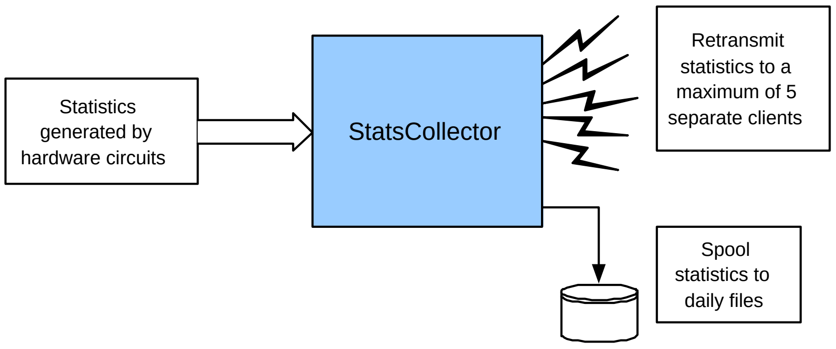

Statistics generated by hardware circuits
StatsCollector
Retransmit statistics to a maximum of 5 separate clients
Spool statistics to daily files

Figure 45: StatsCollector Operational Summary

The statistics files generated by the StatsCollector application have a specific filename format shown below:

`stats%%&&_YYYYMMDD`

> `%%` represents a one-byte (two-hex-digit) number indicating the statistics type
>
> `&&` represents a one-byte (two-hex-digit) number indicating the statistics identifier
>
> `YYYY` is a 4-digit year
>
> `MM` is a 2-digit month
>
> `DD` is a 2-digit day

These files contain ASCII columns of numbers where each column represents a separate statistics value and each row corresponds to a statistics update packet. This format can be directly imported by programs such as Microsoft Excel and can be graphed utilizing gnuplot. A sample perl script is shown below which generates a separate graph of active flows for each daily data file in the current directory:

```
#!/usr/bin/perl

# list of stats files
my @file_list = <stats01*>;

# for each file in the directory
foreach $file (@file_list) {

# extract info from filename
  my $type = substr($file, 5, 2);
  my $id = substr($file, 7, 2);
  my $year = substr($file, 10, 4);
  my $mon = substr($file, 14, 2);
  my $day = substr($file, 16, 2);

# generate gnuplot control file to create chart
  open handle, "> tmp";
  select handle;
  print "set title \"Traffic Stats for $mon/$day/$year\"\n";
  print "set output \"total_flows_$year$mon$day.eps\"\n";
  print "set data style lines\n";
  print "set terminal postscript eps\n";
  print "set xlabel \"Time of day\"\n";
  print "set ylabel \"Total Active Flows\"\n";
  print "set key left\n";
  print "set xdata time\n";
  print "set timefmt \"%H:%M:%S\"\n";
  print "set xrange [\"00:00:00\":\"23:59:59\"]\n";
  print "set format x \"%H:%M\"\n";
  print "\n";
  print "plot \"$file\" using 1:13 title "Active Flows\n";
  close handle;

# execute gnuplot command to generate chart
  system "gnuplot tmp";
# end loop
}
```

9.2 StatApp

The StatApp is a Java application created by Mike Attig (mea1@arl.wustl.edu) to perform real-time graphing. The original StatApp program was modified to accept data generated in the standard statistics format and chart the data in real-time. The following commands can be used to compile and run the StatApp charting application. The port# should be set to the UDP port number of the statistics packets, currently 6501.

```
$ javac *.java
$ java StatApp <port#>
```

Figure 46 displays the output of the StatApp charting various statistics counters from the TCP-Processor. A maximum of 10 data items can be charted in parallel. The various signals are annotated in this diagram for easier reading. The source for the StatApp application can be found at the following web site: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/statapp.html`.

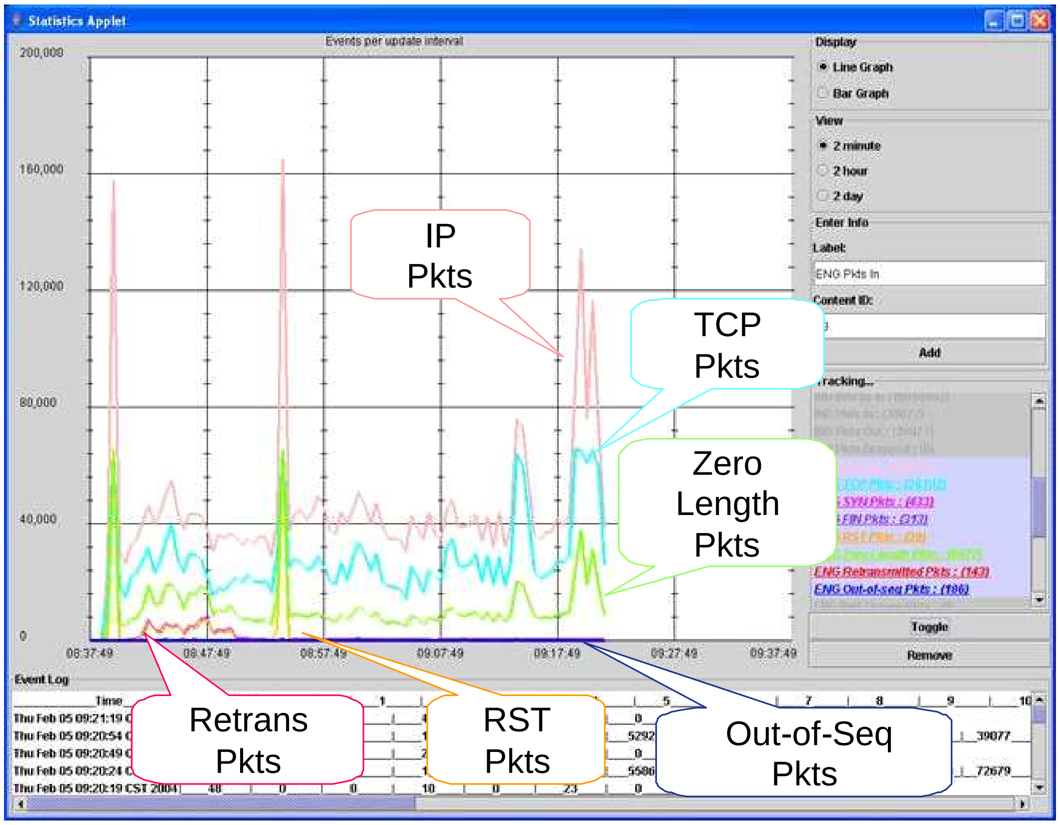
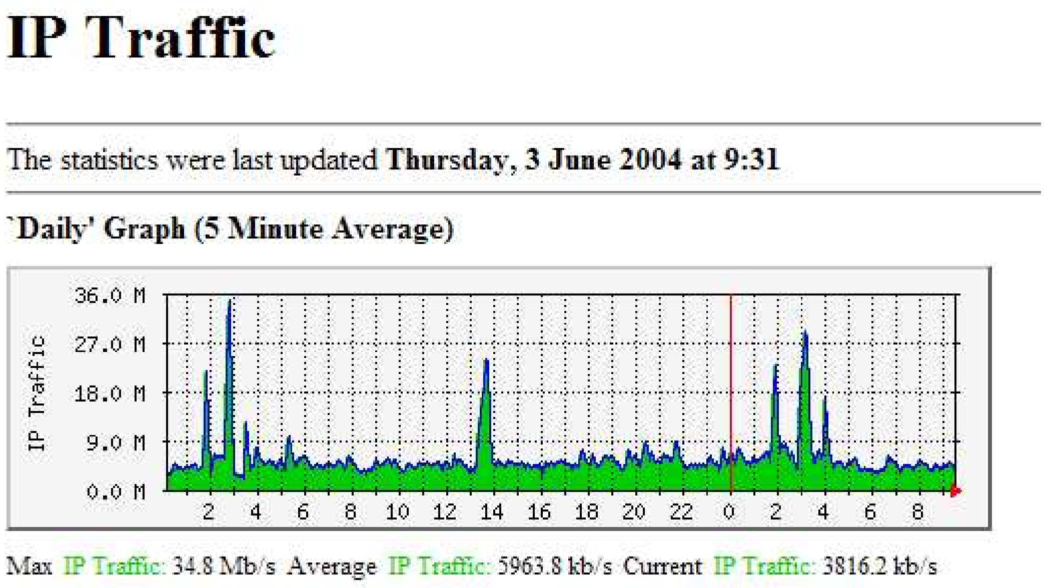


Figure 46: Sample StatApp Chart

9.3 SNMP Support

In order to aid in the dissemination of the statistical information gathered in the various hardware circuits, a Simple Network Management Protocol (SNMP) agent has been developed. This agent listens for the statistics packets generated by the various hardware circuits, maintains internal counters for these statistics, and re-exposes this data as SNMP Management Information Base (MIB) variables. This agent was designed to interface with net-snmp v5.1.1 (`http://net-snmp.sourceforge.net/`).

The distribution for the SNMP agent along with the MIB declaration can be found at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source /gv_snmp_agent_v2.zip`.



IP Traffic

The statistics were last updated **Thursday, 3 June 2004 at 9:31**

`Daily' Graph (5 Minute Average)

Max IP Traffic: 34.8 Mb/s Average IP Traffic: 5963.8 kb/s Current IP Traffic: 3816.2 kb/s

Figure 47: Sample MRTG Generated Chart

9.4 MRTG Charting

The Multi Router Traffic Grapher (MRTG) tool was developed to monitor traffic loads on network links. MRTG generates Hyper Text Markup Language (HTML) web pages on a periodic basis by polling pre-configured SNMP MIB variables. PNG images are generated which provide a visual representation of the traffic loads. This tool has been used to generate graphs of the statistical information maintained by the previously mentioned SNMP agent. A sample chart is shown in Figure 47.

The MRTG charting of statistical information generated by the various hardware circuits was accomplished utilizing the following versions of MRTG and dependent libraries: MRTG v2.10.13, libpng v1.2.5, gd v1.0.22, and zlib v1.1.4. Additional information regarding MRTG can be found at the following web site: `http://www.mrtg.org/`. The MRTG configuration file utilized to chart the statistics information is located at the following URL:
`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source/`
`mrtg_v2.zip`.

References

[1] IETF, “RFC793: Transmission Control Protocol.” http://www.faqs.org/rfcs/rfc793.html, Sep 1981.

[2] D. V. Schuehler and J. Lockwood, “TCP-Splitter: Design, implementation, and operation,” Tech. Rep. WUCSE-2003-14, Washington University in Saint Louis, Mar. 2003.

[3] D. E. Taylor, J. W. Lockwood, and N. Naufel, “Rad module infrastructure of the field-programmable port extender (fpx),” Tech. Rep. WUCS-TM-01-16, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, July 2001.

[4] S. Dharmapurikar and J. W. Lockwood, “Synthesizable design of a multi-module memory controller,” Tech. Rep. WUCS-TM-01-26, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, October 2001.

[5] J. Turner, “Applied Research Laboratory.” http://www.arl.wustl.edu, 2004.

[6] D. E. Taylor, J. W. Lockwood, and S. Dharmapurikar, “Generalized RAD module interface specification of the field-programmable port extender (fpx),” Tech. Rep. WUCS-TM-01-15, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, July 2001. Available on-line as `http://www.arl.wustl.edu/arl/projects/fpx/wugs.ps`.

[7] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke, “Design of a Gigabit ATM Switch,” in *In Proceedings of Infocom 97*, Mar. 1997.

[8] J. D. DeHart, W. D. Richard, E. W. Spitznagel, and D. E. Taylor, “The smart port card: An embedded Unix processor architecture for network management and active networking,” Tech. Rep. WUCS-01-18, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, August 2001.

[9] J. W. Lockwood, “An open platform for development of network processing modules in reprogrammable hardware,” in *IEC DesignCon’01*, (Santa Clara, CA), pp. WB–19, Jan. 2001.

[10] J. W. Lockwood, J. S. Turner, and D. E. Taylor, “Field programmable port extender (FPX) for distributed routing and queuing,” in *ACM International Symposium on Field Programmable Gate Arrays (FPGA’2000)*, (Monterey, CA, USA), pp. 137–144, Feb. 2000.

[11] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, “Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX),” in *ACM International Symposium on Field Programmable Gate Arrays (FPGA’2001)*, (Monterey, CA, USA), pp. 87–93, Feb. 2001.

[12] D. E. T. Todd Sproull, John W. Lockwood, “Control and Configuration Software for a reconfigurable Networking Hardware Platform,” in *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, (Napa, CA), Apr. 2002.

[13] F. Braun, J. Lockwood, and M. Waldvogel, “Reconfigurable router modules using network protocol wrappers,” in *Proceedings of Field-Programmable Logic and Applications*, (Belfast, Northern Ireland), pp. 254–263, Aug. 2001.

[14] F. Braun, J. W. Lockwood, and M. Waldvogel, “Layered protocol wrappers for Internet packet processing in reconfigurable hardware,” in *Proceedings of Symposium on High Performance Interconnects (HotI’01)*, (Stanford, CA, USA), pp. 93–98, Aug. 2001.

[15] S. McCanne, C. Leres, and V. Jacobson, “tcpdump.” ftp://ftp.ee.lbl.gov, 1998.

[16] F. Braun, J. W. Lockwood, and M. Waldvogel, “Layered protocol wrappers for Internet packet processing in reconfigurable hardware,” Tech. Rep. WU-CS-01-10, Washington University in Saint Louis, Department of Computer Science, June 2001.