Report Number: WUCSE-2003-14

2003-03-18

TCP-Splitter: Design, Implementation and Operation

David V. Schuehler and John Lockwood

TCP-Splitter is a hardware circuit which facilitates the monitoring of TCP/IP data streams. When located within high-speed networking equipment, this circuit provides ordered TCP byte streams for all TCP flows at line rates. This document provides and in-depth look at the design and implementation of the TCP-Splitter circuit. The operation of the TCP-Splitter with three sample client applications is also described.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-14

TCP-Splitter: Design, Implementation and Operation

Authors: David V. Schuehler, John Lockwood

March 18, 2003

Abstract: TCP-Splitter is a hardware circuit which facilitates the monitoring of TCP/IP data streams. When located within high-speed networking equipment, this circuit provides ordered TCP byte streams for all TCP flows at line rates. This document provides and in-depth look at the design and implementation of the TCP-Splitter circuit. The operation of the TCP-Splitter with three sample client applications is also described.

TCP-Splitter: Design, Implementation and Operation

David V. Schuehler and John Lockwood

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

WUCSE-2003-14
http://www.arl.wustl.edu/arl/projects/fpx

March 18, 2003

Abstract

TCP-Splitter is a hardware circuit which facilitates the monitoring of TCP/IP data streams. When located within high-speed networking equipment, this circuit provides ordered TCP byte streams for all TCP flows at line rates. This document provides and in-depth look at the design and implementation of the TCP-Splitter circuit. The operation of the TCP-Splitter with three sample client applications is also described.

1 INTRODUCTION

The TCP-Splitter was designed to provide for the monitoring of data transmitted through high-speed networking equipment. Most of the Internet traffic today is based on the Transmission Control Protocol (TCP) / Internet Protocol (IP) [1] suite of protocols. By performing flow classification and stream re-ordering tasks, the TCP-Splitter circuit eases the processing burden of the client applications, allowing them to concentrate on application specific logic. The TCP-Splitter circuit is described using the VLSI Hardware Description Language (VHDL) for use in Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs). This facilitates the processing of network packets at high data rates. TCP-Splitter was first introduced at the 2002 Hot Interconnects conference [2].

This paper will be broken down into five sections. This first section will discuss the background of the TCP-Splitter project. The second section covers the hardware environment in which the TCP-Splitter circuit was originally designed and tested. The third section discusses the details of the TCP-Splitter design and implementation. The fourth section examines a companion circuit to the TCP-Splitter called *Defragment*. The fifth section describes the operation of three different test applications that use the TCP-Splitter circuit.

2 Background

The original goal of this project was to develop a hardware circuit capable of working with TCP/IP data streams in a high speed networking environment. Initial considerations included the development of a full TCP/IP protocol stack in hardware. This course of action was discounted for the following reasons:

- A full TCP stack implementation would act as a connection endpoint, which would terminate connections in the interior of the network.
- The number of simultaneously connections that could be supported by a hardware implementation would be small.
- The complexity of the TCP protocol stack prevents a lightweight solution from being developed.

The direction of the project turned to one of flow monitoring instead of protocol termination. The development of a purely passive TCP flow monitor was discussed and discarded due to the potentially large memory requirement needed for reassembly buffers. The maximum window scale factor supported by the TCP protocol is 2^{14} bytes [3]. A TCP flow using this window size would potentially require 1 GByte of

memory for use as a reassembly buffer. Even with much more reasonable windows sizes, the total number of simultaneously monitored connections would be limited.

The TCP-Splitter project finally settled on a goal to develop a lightweight hardware circuit capable of monitoring large numbers of TCP flows at OC-48 (2.4Gbps) line rates. In order to meet these goals within the practical bounds of of today's hardware, a non-passive monitoring solution was developed. The TCP-Splitter circuit actively drops selected out-of-order TCP data packets in order to ensure an in-order flow of data through the circuit. This alleviates the circuit from maintaining large reassembly buffers that would otherwise be required to reorder the monitored data packets.

3 Environment

The TCP-Splitter, developed at the Washington University Applied Research Laboratory (ARL), leverages previous research performed at this same facility. The Washington University Gigabit Switch (WUGS) and the Field-Programmable Port Extender (FPX) plug-in module were used as a test bed for the TCP-SPlitter. The Layered Protocol Wrappers, also developed at the ARL, provide a communications framework in which the TCP-Splitter circuit sits.

3.1 Washington University Gigabit Switch

The Washington University Gigabit Switch [4] was developed as a platform upon which high speed networking research could take place. The WUGS, as shown in figure 1, contains eight I/O ports connected to a switching backplane. All data paths on the switch are 32 bits wide and the system is clocked at 62.5MHz. A throughput of 20 Gb/s can be achieved with this network switch. The switching fabric operates fast enough to drive each of the eight I/O ports at a data rate of 2.4 Gb/s.

Each of the I/O ports supports various adapter cards. Currently, there exists adapter cards for a Gigabit Ethernet line card, a Gigabit ATM line card, an OC3 ATM line card, a Smart Port Card (SPC), a Smart Port Card II (SPC2), and a Field-programmable Port Extender (FPX). The SPC and SPC2 contain a Pentium processor and Pentium III processor respectively that can both be programmed over the network. The FPX card contains large Xilinx Field Programmable Gate Arrays (FPGAs) that can be remotely reconfigured.

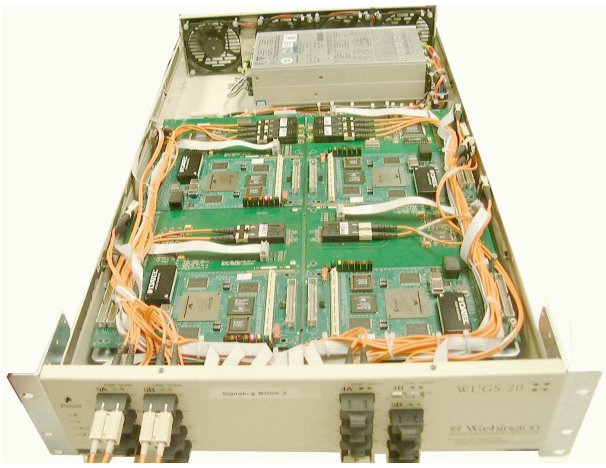

Figure 1: Washington University Gigabit Switch Loaded with Four FPX Devices

3.2 Field Programmable Port Extender

The Field Programmable Port Extender (FPX) [5] is a modular switch component for the WUGS. All logic on the FPX card is contained in reprogrammable FPGAs. The board contains two Xilinx FPGAs, the Network Interface Device (NID) and the Reprogrammable Applications Device (RAD). The RAD contains interfaces to off-chip Synchronous Dynamic RAM (SDRAM) and Zero Bus Turnaround (ZBT) pipelined Static Random Access Memory (SRAM). The FPX device is capable of communicating at speeds of 2.4 gigabits per second (OC-48). The RAD device can be remotely reprogrammed by sending specially configured control cells to the device. A diagram of the components and data flow of the FPX module can be seen in figure 2 [6].

Remote configuration and control of the FPX is supported by a suite of tools called NCHARGE (Networked Configurable Hardware Administrator for Reconfiguration and Governing via End-systems) [7]. 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 modules using standard TCP/IP sockets.

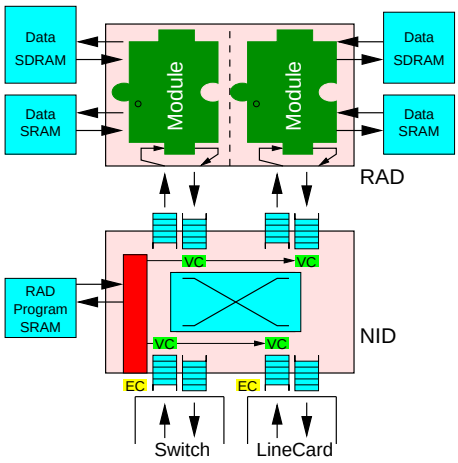

Figure 2: Field Programmable Port Extender

3.3 Protocol Wrappers

Internet packets are processed on the FPX card using a set of Layered Protocol Wrappers [8]. These protocol wrappers were developed to provide a mechanism for processing the User Datagram Protocol (UDP) within a hardware circuit. The processing of each protocol layer is isolated in an independent protocol wrapper. By stacking the wrappers, higher level protocols can be processed without concern for the underlying transport level protocols. The Layered Protocol Wrappers are shown in figure 3. Ingress data is processed by successive layers of the protocol stack. The output of the UDP wrapper consist of data packets that are passed to a client application for processing. Outbound data packets generated by a client application are passed back through the various protocol layers for transmission to an end client.

4 Design

The TCP-Splitter circuit was implemented using VHDL. This allows the TCP-Splitter implementation to easily be incorporated into almost any FPGA or ASIC hardware design. The name TCP-Splitter comes from the concept that a TCP flow is split in two, with one copy of the TCP byte stream continuing on towards the end host and another copy of the byte stream passing on to a client hardware circuit for additional processing.

The TCP-Splitter circuit interface supports the specification of a generic parameter to indicate whether

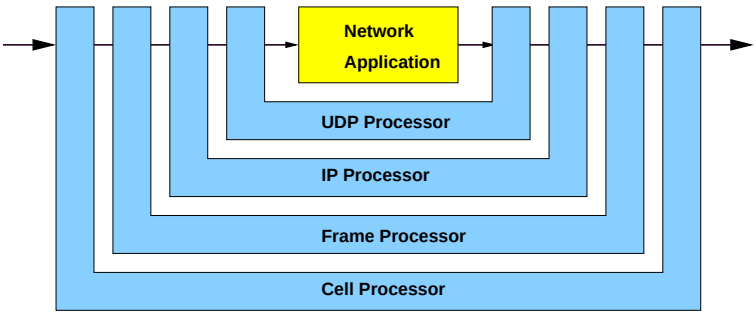

Figure 3: Layered Protocol Wrappers

or to operate in simulation mode. When this parameter is set to one, the TCP-Splitter does not initialize SRAM on startup and maps all flows to the hash table entry located at SRAM address zero. This allows simulations of the circuit to be performed without having to wait 256 thousand clock cycles for SRAM to be initialized before sending in the first packet. This parameter should be set when simulating the circuit and cleared when building a design to operate in hardware.

4.1 Framework

The TCP-Splitter expands upon the Layered Protocol Wrappers work to include support for processing the TCP protocol. The TCP-Splitter circuit replaces the UDP Wrapper and connects directly to the IP Wrapper as shown in figure 4. A client application sits on top of the TCP-Splitter circuit. As TCP/IP packets enter the network switch, they are routed through the successive layers of the Layered Protocol Wrappers. The TCP-Splitter splits off an ordered TCP data stream and passes it onto a client application. The TCP packet is also passed to the outbound Layered Protocol Wrappers for transmission toward the packet destination. Since the client application does not sit directly in the network path, packet throughput is not affected by the complexity of the application processing.

4.2 Internals

The TCP-Splitter itself is internally broken up into separate processing modules and containers. The file `tcpsplitter.vhd` implements a container for the external interfaces to support the transfer of inbound data, outbound data, client data, and communication with static RAM (SRAM). This container internally

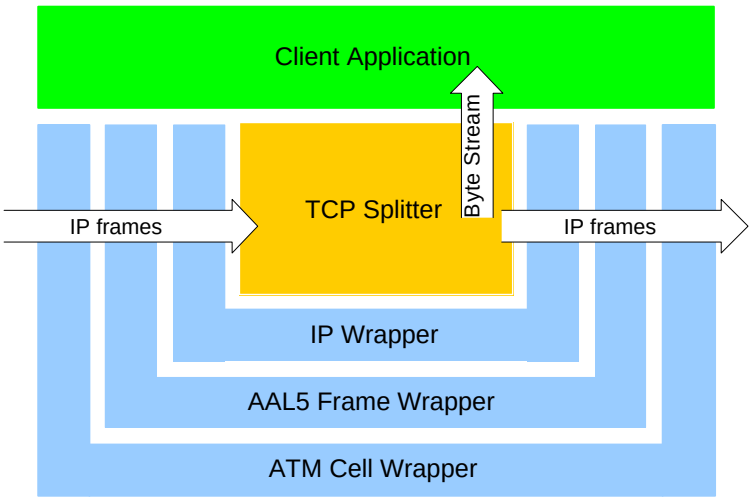

Figure 4: TCP-Splitter Data Flow

routes signals between the various external interfaces of the IP Wrapper and the *TCPProc* processing engine.

The main ingress and egress data interfaces utilized by TCP-Splitter contain a 32 bit wide data path. On every rising clock edge, another data word is processed. In addition to the data word, a collection of control signals are also passed between the IP Wrapper and the TCP-Splitter. These signals include a start of frame (SOF) signal, a start of payload (SOP) signal, and an end of frame (EOF) signal. A transmit cell available (TCA) signal is passed in the opposite direction and is used as a flow control mechanism.

The TCP-Splitter is internally composed of two main processing components called *TCPInput* and *TCPOutput*. The `tcpproc.vhd` file implements a container which houses these two components. The external interface comprises inbound data, outbound data, client data, and a memory interface. The inbound data and memory interface are routed to the *TCPInput* module. Output signals from the *TCPInput* module are routed to the *TCPOutput* module along with signals for outbound and client data. The overall layout of the circuit can be seen in figure 5.

4.3 Input Processing

The *TCPInput* module contains most of the logic associated with the TCP-Splitter circuit and also performs most of the work. Inbound packets are validated, checksummed, and classified before being passed on to the

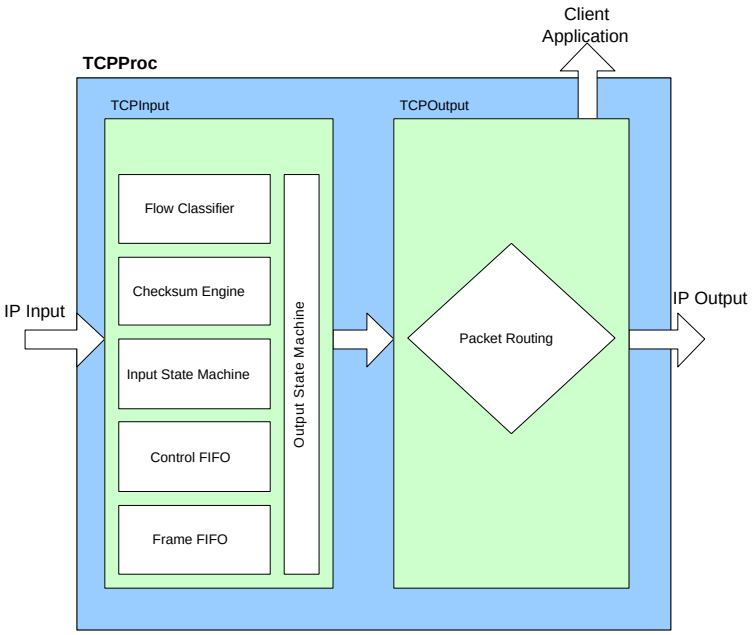

Figure 5: TCP-Splitter Internals

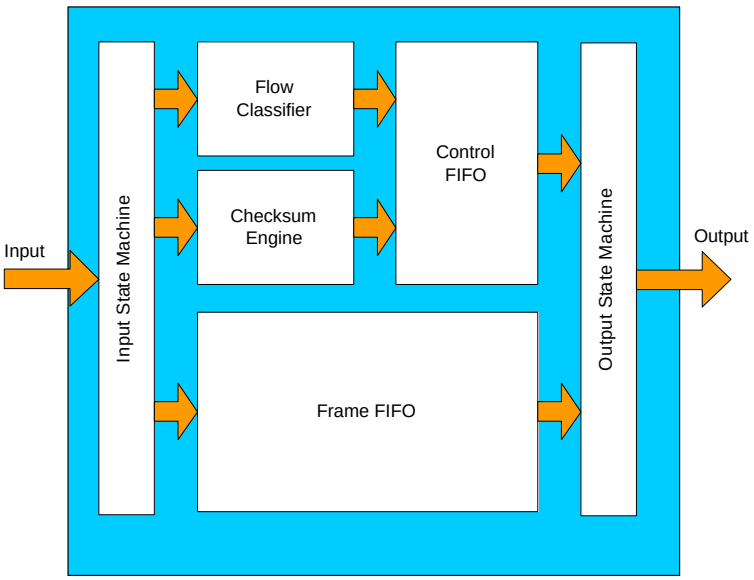

Figure 6: TCP-Splitter Inbound Processing

TCPOutput module. The implementation of the *TCPInput* module is contained within the `tcpinbound.vhd` file and supports an external interface for packet ingress, packet egress, and communication with SRAM. The processing of data can be subdivided into six logical components: an Input State Machine, a Flow Classifier, a Checksum Engine, an Output State Machine, a Control FIFO and a Frame FIFO. Although there is no distinction made between these components within the source VHDL code, it is easier to understand the operation of the TCP-Splitter in this manner. The interaction of these six components can be seen in figure 6.

As data enters the *TCPInput* module, it is first processed by the Input State Machine which determines the current packet processing state. This state information is utilized by the other components of this module when processing data.

The Flow Classifier performs a simple XOR hash of the source/destination IP addresses and the source/destination TCP ports. The source and destination values are bit shifted so that packets travelling in the opposite direction of the same flow hash to different locations in the classification table. This hash value is used as a direct index into the flow classification table contained in SRAM. Each SRAM address corresponds to a 36 bit data field. Currently, 33 bits of each entry are used for the flow table. The low order 32 bits contain the

next expected sequence number for a particular flow. The 33^{rd} bit indicates whether or not a flow is active, with a value of zero corresponding to an inactive flow. The Flow Classifier also writes updates to SRAM indicating the next expected sequence number and the current flow state (active or inactive).

The Checksum Engine checks the protocol field of the IP header to determine whether or not the current packet is a TCP packet. If the packet is a TCP packet, then the TCP checksum is computed and validated against the packet being processed. The checksum verification result is used when deciding how to route packets.

The results of the Checksum Engine and Flow Classifier are written to the Control FIFO. The purpose of the Control FIFO is to hold control information while preceding frames are still clocked out of the Frame FIFO. The Frame FIFO buffers packets so that the results of the Checksum and Flow Classification operations can be delivered to the *TCPOutput* module along with the beginning of a packet.

The Output State Machine is responsible for reading data from the Control FIFO and Frame FIFO and passes this data to the *TCPOutput* module. On detection of a non-empty control FIFO, the Output State Machine changes state and reads control information for a single packet from the control FIFO. Next, a state is entered where packet data is clocked out of the frame FIFO and is passed to the output processing section. After reading an entire packet from the frame FIFO, the Output State Machine returns to its initial state waiting for a non-empty control FIFO.

4.4 Output Processing

The output processing logic is contained within the `tcpoutbound.vhd` source file. This module is responsible for packet routing and delivering data to the appropriate output interface. The Output processing engine receives control signals and packet data from the *TCPInput* module. These signals are listed as follows:

- a TCP data enable signal
- a 2 bit signal indicating which data bytes are valid
- a TCP packet indication signal
- a valid checksum signal
- a N bit flow identifier (currently 18 bits)
- a new flow indication signal

- a signal indicating that the packet should be forwarded
- a sequence number match signal
- a signal indicating valid data
- an end of flow indication signal

Based on the values of these control signals, the *TCPOutput* module routes packets utilizing one of three different routing options. Packets can either be (1) passed to the outbound IP Wrapper only, (2) passed to both the outbound IP Wrapper and to the Client Application, or (3) dropped. The rules for making this routing decision are as follows:

- All non TCP packets are sent to the outbound IP stack. Since the IP Wrapper passes all IP packets to the TCP-Splitter, non TCP protocol packets need to forwarded to their respective destinations.
- All TCP packets with invalid checksums are dropped. The logic here is that since the packet is invalid, there is no sense in cluttering the network with data that will be dropped at the end destination.
- All TCP packets with sequence numbers less than or equal to the current expected sequence number, are sent to the outbound IP stack. This signifies a TCP data packet that has already been processed by the TCP-Splitter and usually coincides with the a retransmission of a packet that was dropped between TCP-Splitter and the end destination.
- All TCP packets with sequence numbers greater than the current expected sequence number are dropped. The TCP-Splitter drops these out of order packets in order to eliminate the need for large reassembly buffers. TCP flows passing through this section of logic must pass in order.
- All TCP SYN packets are sent to the outbound IP stack. These packets are associated with the three way handshake required to establish a TCP connections. These packets contain no user data and the TCP-Splitter is able to conserve resources by not processing these packets.
- All other packets are forwarded both to the outbound IP stack and the client application. These packets correspond to in TCP data packets which have arrived in order in relation to their respective flows.

This set of packet routing rules ensures that only in-order data streams are passed to the client application. The retransmission of missing data is triggered by dropping all packets with sequence numbers greater than the next expected sequence number. After processing by the TCP-Splitter circuit. the destination host will only receive packets with sequence numbers equal to or less than the next expected sequence number.

4.5 Client Interface

The Client Interface was designed to provide a simple mechanism for delivering a large number of TCP streams to a client application. Lower layer protocol headers are also transmitted to the client via this same mechanism in order to provide the client application with access to the various header fields. This eliminates the need for a complex interface which would be required if the client had to issue queries to the TCP-Splitter to retrieve information contained in these fields. This also saves the TCP-Splitter from having to cache this information.

The client interface contains 11 signals which contain data and control information. All signals are active high, meaning that when the voltage of the signal is equal to Vcc, then the signal represents a boolean true or value of 1.

D_OUT_APPL a 32 bit data word

DataEn_OUT_APPL a 1 bit data enable signal indicating that the data word contains valid data

SOF_OUT_APPL a 1 bit start of frame signal - the IP protocol header begins on the next clock cycle

EOF_OUT_APPL a 1 bit end of frame signal - indicating the last payload word of a packet - two words of AAL5 trailer will follow this signal

SOP_OUT_APPL a 1 bit start of payload signal - indicating the first word of the TCP header

TDE_OUT_APPL a 1 bit TCP data enable signal - indicates valid TCP payload data can be read from the data word

BYTS_OUT_APPL a 2 bit valid bytes vector - indicating the number of valid TCP data bytes contained in the data word

FLOWID_OUT_APPL a 18 bit vector - indicates the identifier to associated with this TCP data flow

NEWFLOW_OUT_APPL a 1 bit signal - signifies the start of a new flow

ENDFLOW_OUT_APPL a 1 bit signal - signifies the end of an existing flow

A flow control signal called transmit cell available (TCA) is also included in the interface, but has not been fully implemented. The current stumbling point is whether or not packets should be dropped while the client application asserts the TCA signal. Currently the TCA signal is passed on to the inbound Layered Protocol Wrappers. Since the TCP-Splitter buffers packets, a client application should expect to receive up to 1500 bytes of data after asserting the TCA signal. The issue of flow control will probably be addressed in the next version of the TCP-Splitter.

In order to help better illustrate the client interface, a series of four wave form diagrams are included which depict the four main phases of flow processing. The four types of packets that a client will process are (1) the start of a new flow, (2) additional data related to an existing flow, (3) a packet with no stream data, and (4) the end or termination of an existing flow.

The first waveform deals with the processing of a new flow as shown in figure 7. The arrival of a packet is signified by the SOF signal going high for one clock cycle. A packet is to be considered the start of a new flow if the NEWFLOW signal is also high at this time. The NEWFLOW signal will remain high for the duration of the packet. As with all packets received through the client interface, the FLOWID will be valid from the assertion of SOF through to the assertion of the EOF signal. Immediately after receiving the SOF signal, the IP protocol header will be clocked into the client application 4 bytes at a time. The start of the TCP header will be signified by the assertion of the SOP signal. Valid TCP stream data can be read from the data word when the TDE signal is asserted. The number of valid data bytes contained in the data word is described by the BYTES bit vector. A value of 00_2 corresponds to one valid data byte contained in the high order 8 bits of the data word. Values of 01_2, 10_2, and 11_2 correspond to two, three, and four valid data bytes respectively. The end of the packet is signalled by the assertion of the EOF signal.

The second TCP-Splitter waveform corresponds to the reception of additional stream data associated with an existing flow as shown in figure 8. The start of packet is once again signalled by the assertion of the SOF signal. During this period, the NEWFLOW and ENDFLOW signals will both remain low and the FLOWID field will be valid. Following this signal, the IP header will be clocked into the client application. The start of the TCP header will coincide with the assertion of the SOP signal. Following the protocol headers will be the TCP stream data. The TDE signal will be asserted while the in-order byte stream is being clocked into the client application. The BYTES bit vector will signify which of the four data bytes contained in the data word are valid. The end of the packet is signalled by the assertion of EOF.

Not all TCP packets contain stream data. When this situation occurs, as in the reception of an acknowledgment packet, the TCP-Splitter will deliver the packet to the client application similar to the case when receiving additional data. The difference is that the TDE signal will not be asserted when there is no stream data contained in the packet. A sample wave form describing this occurrence can be seen in figure 9.

The fourth TCP-Splitter waveform corresponds to the reception of an end of flow signal as shown in figure 10. The termination of a TCP flow is triggered by the originating host when either the TCP RST or TCP FIN flag is set in the TCP protocol header. The TCP-Splitter detects this condition and asserts

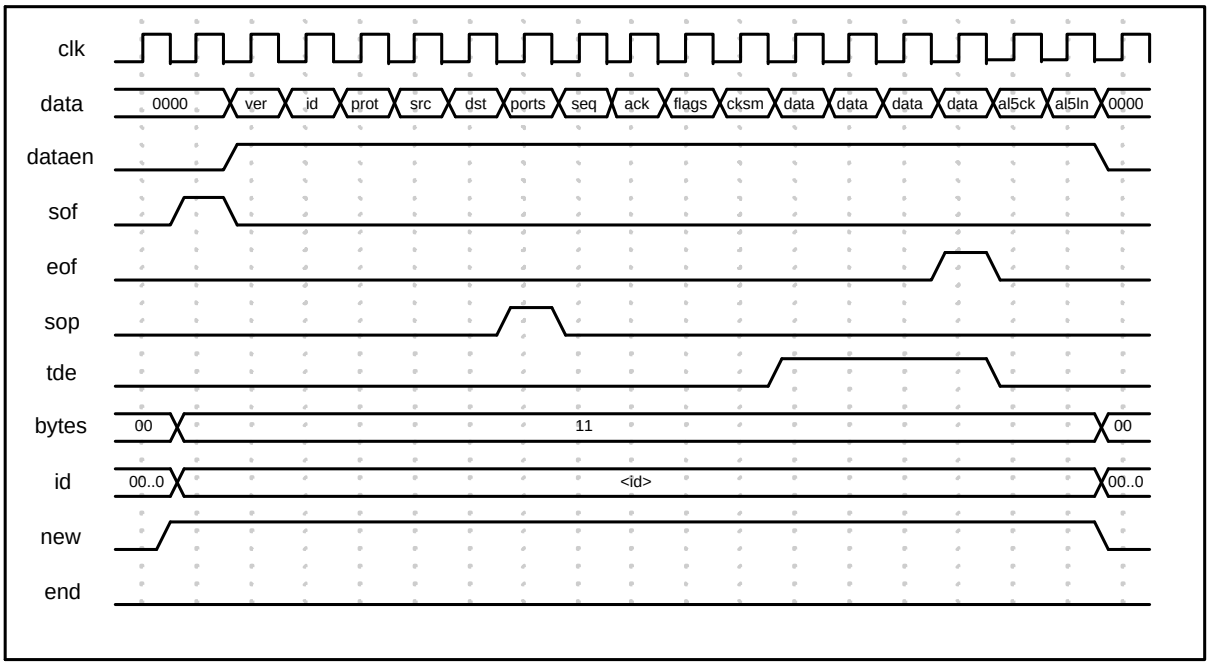

Figure 7: TCP-Splitter New Flow Waveform

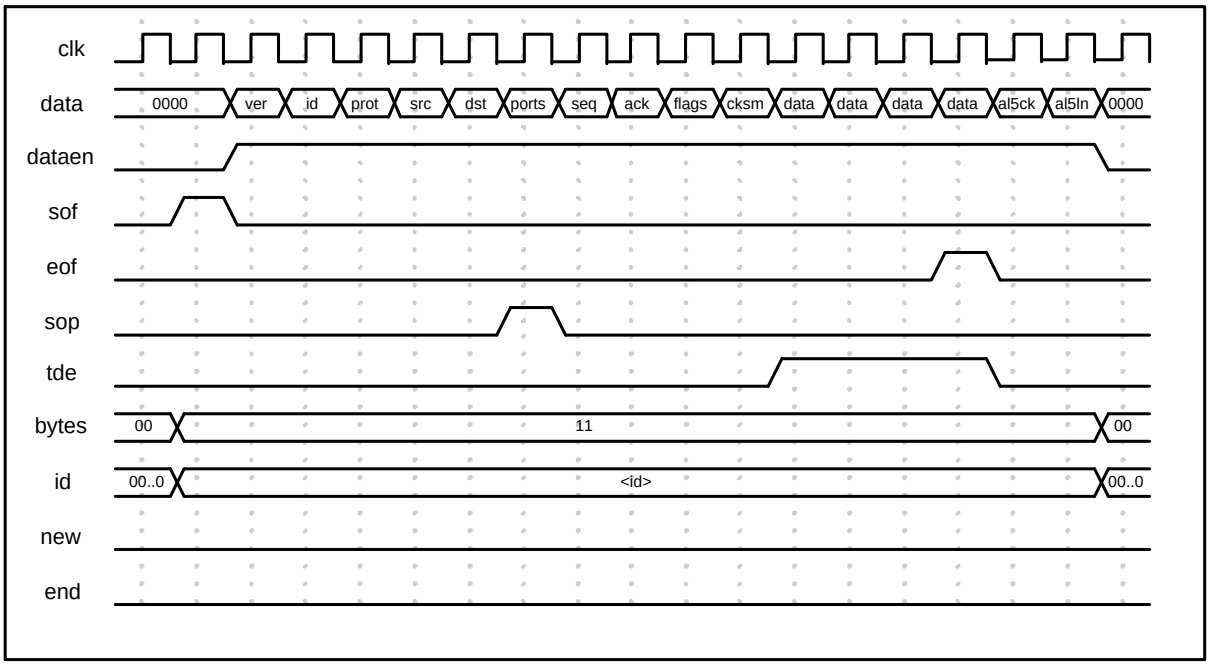

Figure 8: TCP-Splitter Additional Data Waveform

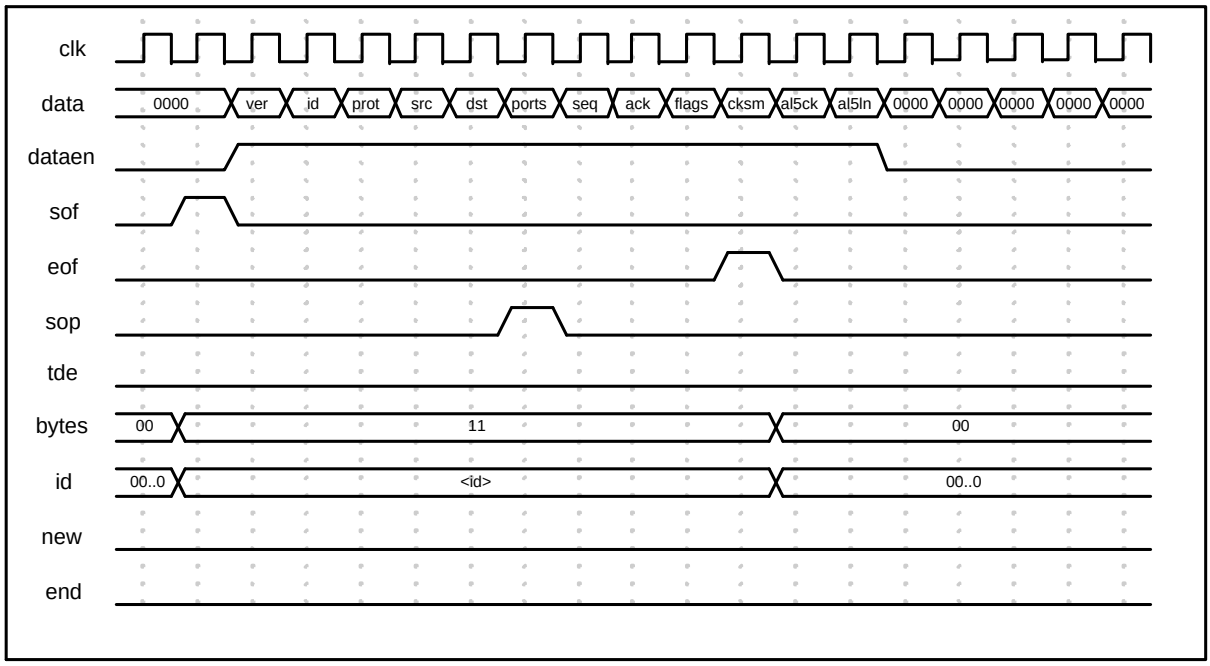

Figure 9: TCP-Splitter No Data Waveform

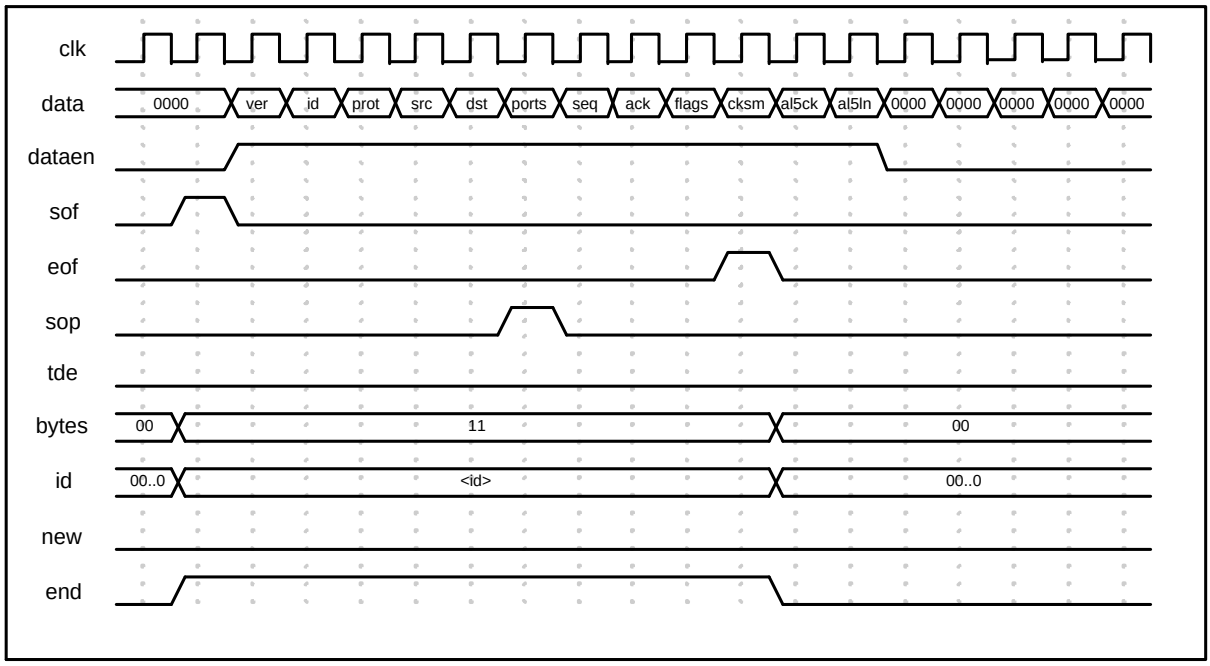

Figure 10: TCP-Splitter End Flow Waveform

the ENDFLOW signal as this packet is passed to the client. Upon reception of the SOF signal, the client application should check both the ENDFLOW and NEWFLOW signals to determine which of the three types of packets is being received. Upon reception of an end of flow indication, the client application should not expect to receive any further data associated with the particular flow and may free any resources utilized in the processing of that flow.

The current TCP-Splitter flow classifier does not prevent collisions of flow identifiers for separate flows. If this occurs, it is possible to receive a start of flow for a currently active flow identifier. That is, two NEWFLOW signals will be received without an intervening ENDFLOW signal. In this case, the client should assume that the previous flow has terminated and that a new flow has started with the same flow identifier.

5 Defragment Circuit

Defragment is a companion circuit which is typically placed between the TCP-Splitter client interface and the client application. The purpose of the *Defragment* circuit is two-fold, first to consolidate TCP stream byte fragments and second to provide simple TCP port filtering.

The *Defragment* circuit was developed after it was realized that several applications all exhibited similar requirements. The internal data bus of the TCP-Splitter is 32-bits wide. When an odd length TCP packet is processed, not all of the data bytes are valid in the final data work of the packet. Many client applications would prefer to process full 32-bit data words and no to have to worry about odd byte lengths, should they occur while processing TCP data packets. An illustration of the byte packing operation performed by the *Defragment* circuit can be seen in figure 11. As data words are clocked out of the TCP-Splitter circuit, the number of valid bytes for each clock cycle corresponds to the TCP packet length. When TCP data packet lengths are not divisible by four, data words clocked into client applications which have non valid data bytes. These odd length data words are compressed and normalized by the *Defragment* circuit and result in a four byte data words being clocked into the client application.

The other purpose of the *Defragment* circuit is to provide a simple TCP port filtering function. This is accomplished by having the client application specify two generic parameters when they instantiate the circuit. The generic parameters specify a source port filter and a destination port filter. Packets entering the *Defragment* circuit with ports matching either of these filters will be passed to the output ports. All other TCP stream data packets will be dropped by the *Defragment* circuit. The implementation of the *Degragment* circuit can be found in the `defragment.vhd` source file.

5.1 Client Interface

The *Defragment* circuit contains a similar client interface to that of the TCP-Splitter but the waveforms generated by the *Defragment* circuit are different. The client interface signals are listed as follows:

DATA_OUT a 32 bit data word

DATAEN_OUT a 1 bit data enable signal indicating that the data word contains valid data

SOF_OUT a 1 bit start of frame signal - the IP protocol header begins on the next clock cycle

EOF_OUT a 1 bit end of frame signal - indicating the last payload word of a packet - two words of AAL5 trailer will follow this signal

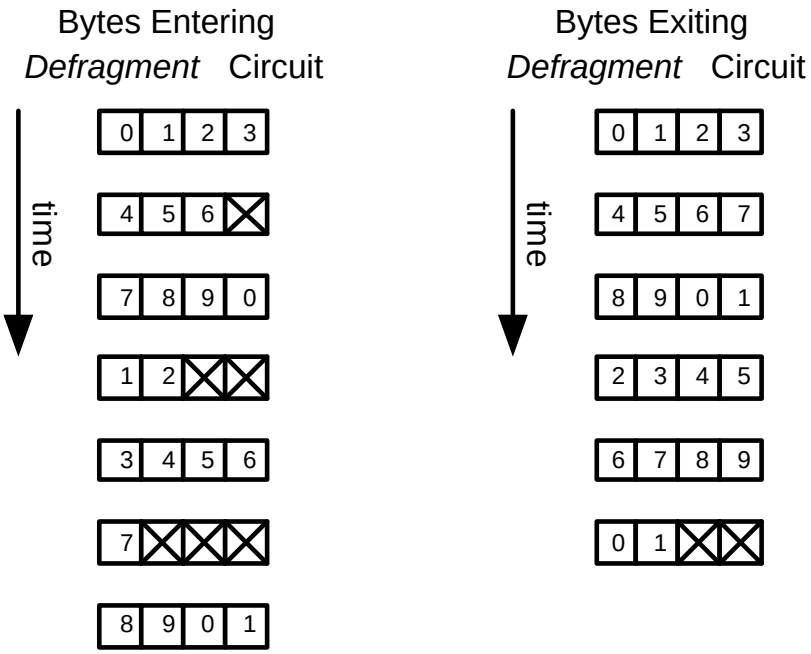

Figure 11: Defragment byte packing operation

SOP_OUT a 1 bit start of payload signal - indicating the first word of the TCP header

TDE_OUT a 1 bit TCP data enable signal - indicates valid TCP payload data can be read from the data word

BYTES_OUT a 2 bit valid bytes vector - indicating the number of valid TCP data bytes contained in the data word

FLOWID_OUT a 18 bit vector - indicates the identifier to associated with this TCP data flow

NEWFLOW_OUT a 1 bit signal - signifies the start of a new flow

ENDFLOW_OUT a 1 bit signal - signifies the end of an existing flow

A series of wave form diagrams were produced to depict the external interface and client interaction with the *Defragment* circuit. The following four wave forms illustrate (1) the start of a new flow, (2) additional data related to an existing flow, (3) a packet with does not match the specified port filter, and (4) the end or termination of an existing flow.

The first waveform deals with the processing of a new flow as shown in figure 12. Similar to the TCP-Splitter client interface, the arrival of a packet is signified by the SOF signal going high for one clock cycle. Because port filtering is performed by the *Defragment* circuit, the NEWFLOW signal is not asserted until after the TCP ports have been processed. This occurs one clock cycle after the assertion of the SOP signal. The other difference from the operation of the TCP-Splitter client interface is that the valid BYTES bit vector is only enabled when TDE is asserted. All data bytes for protocol header fields will be valid when the DATAEN signal is asserted.

The second *Defragment* waveform corresponds to the reception of additional stream data associated with an existing flow as shown in figure 13. The start of packet is signalled by the assertion of the SOF signal and the FLOWID will contain be valid. During the duration of the packet, the NEWFLOW and ENDFLOW signals will both remain low. The TDE signal will be asserted while valid stream data exists on the data bus. The BYTES bit vector will indicate which of the four data bytes contain valid data.

When packets arrive which do not contain any client data or do not match the port filter criteria, signals will be driven in the client interface in the following manner. Just as for other packets, the SOF, SOP, EOF, DATAEN, and ID signals will be asserted during the processing of one of these packets. The difference is that none of the other signals will indicate that there is data to be processed. In this situation, the client should ignore this packet. A sample wave form of this occurrence can be seen in figure 14.

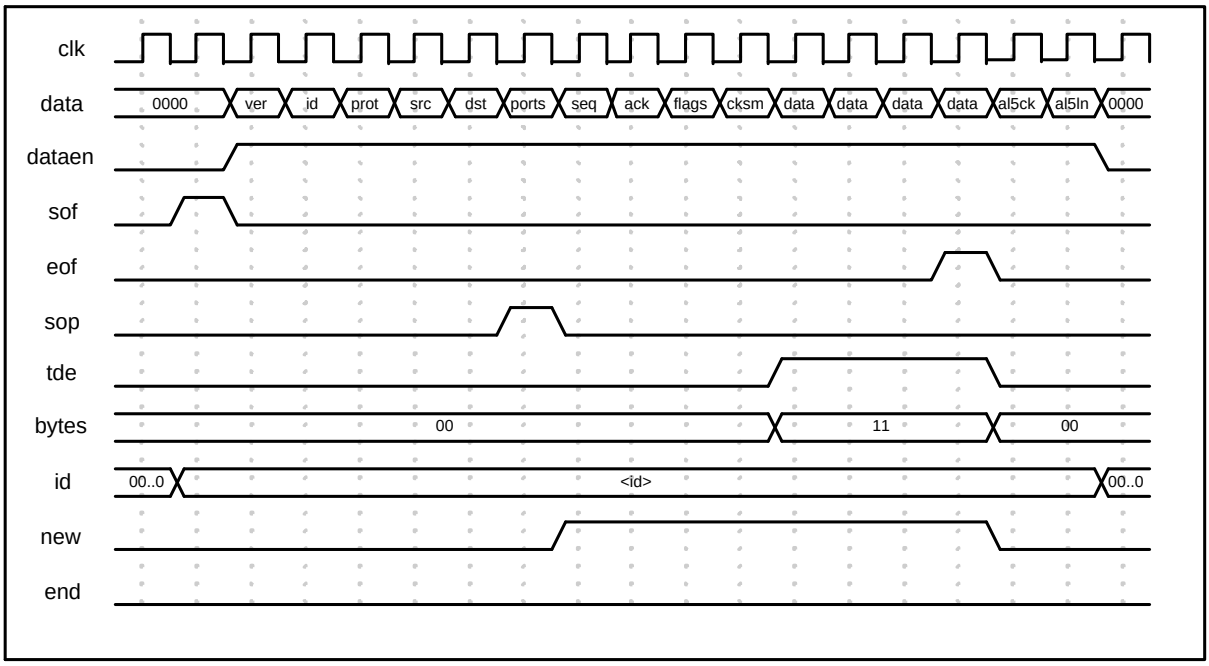

Figure 12: Defragment New Flow Waveform

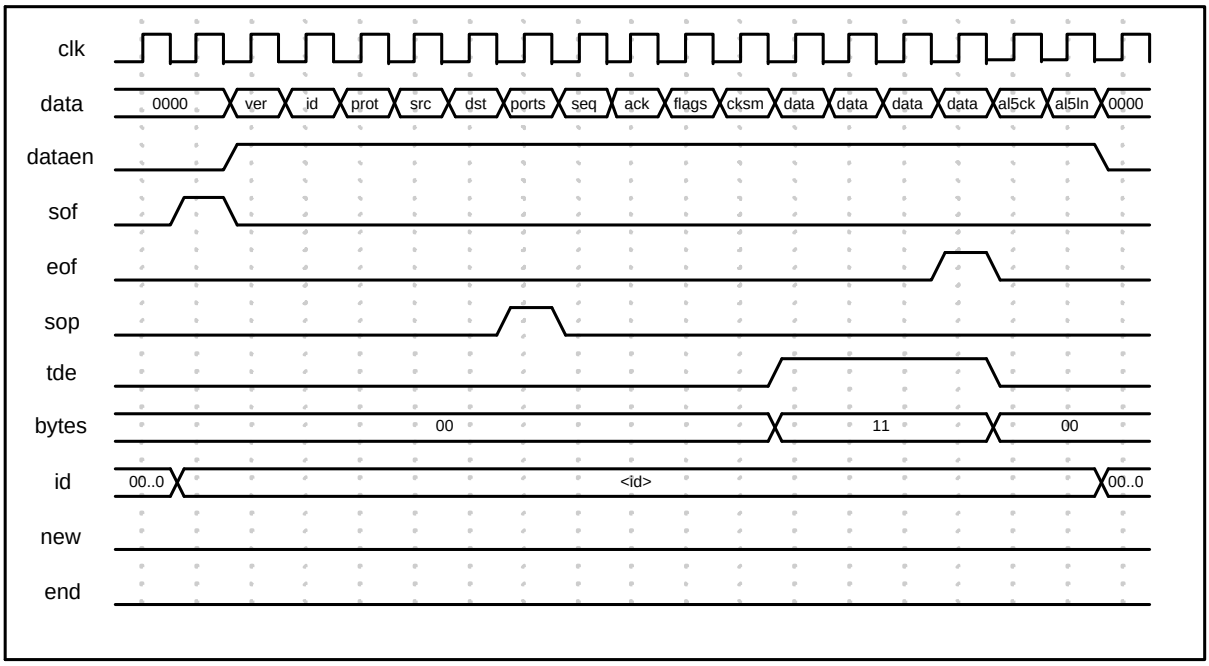

Figure 13: Defragment Additional Data Waveform

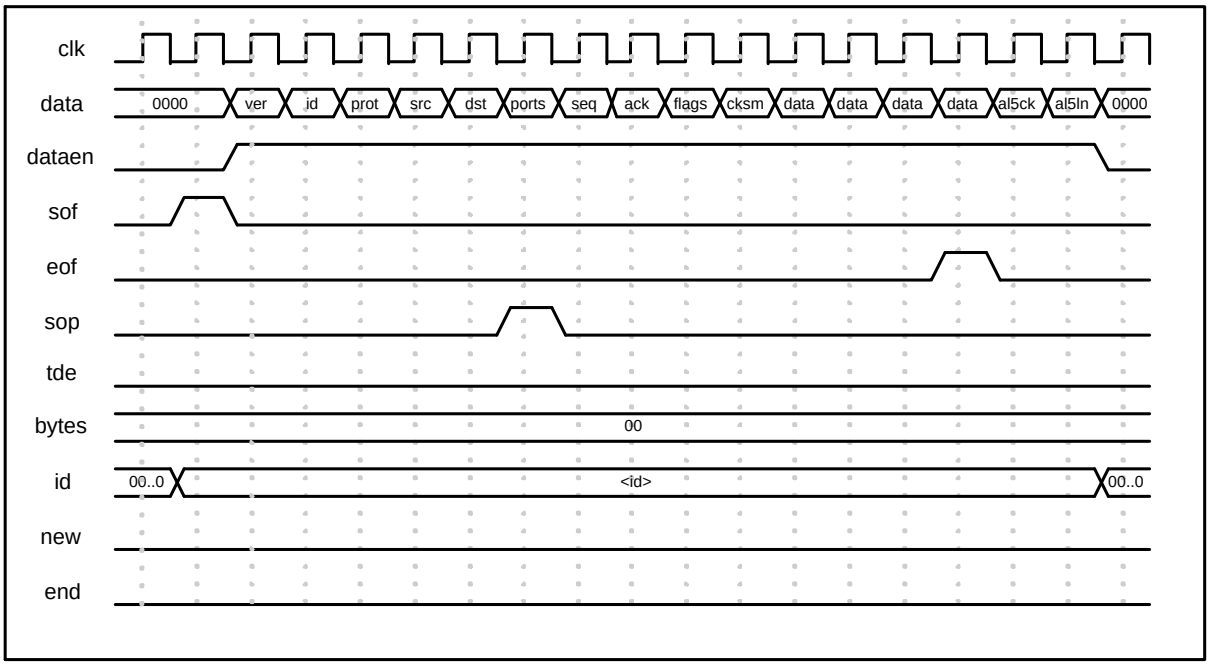

Figure 14: Defragment No Filter Match Waveform

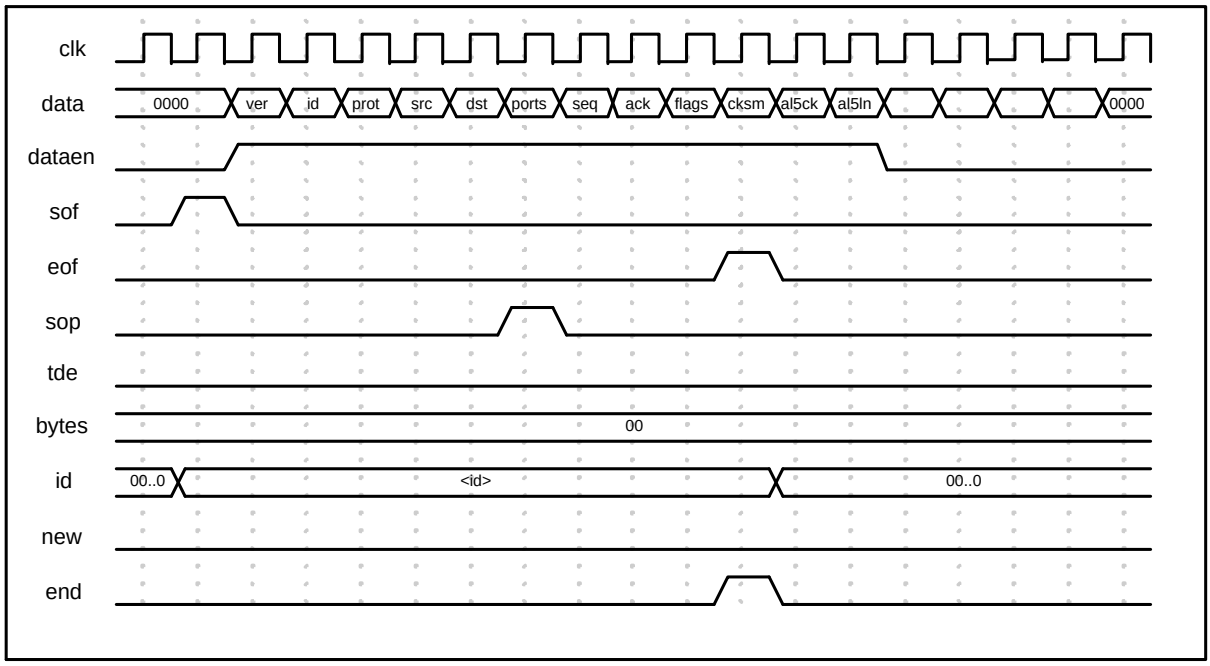

Figure 15: Defragment End Flow Waveform

The fourth *Defragment* waveform corresponds to the indication that a flow has been terminated as shown in figure 15. In this case, the ENDFLOW signal will be asserted in conjunction with the EOF signal for the frame. All other signals will be processed normally. If a data fragment of one to three bytes is contained in the *Defragment* circuit when the flow is terminated, the last fragment of data will be injected into the packet replacing the sequence number of the TCP protocol header. The TDE signal will also be asserted at this time so that the client application can perform normal processing on the last piece of data associated with a particular flow. This ensures that all data is passed to the client before receiving the ENDFLOW signal.

6 Applications

This section covers several of the initial applications developed for the TCP-Splitter technology. These applications were used during the development and debugging phases of the TCP-Splitter project. The ByteCount and TCPClient applications are simple applications meant to show the proper way to interface

with the TCP-Splitter and *Defragment* circuits. The Programmer application is an example of a useful application which utilizes the TCP-Splitter technology. The source code for all of these applications can be found if the cvs code repository for the FPX group in the ARL.

6.1 ByteCount

The ByteCount application was the first TCP-Splitter client application developed. Its main purpose was to provide an environment in which the operation of the TCP-Splitter circuit could be validated. As the name suggests, the ByteCount application counts TCP stream data bytes. After every packet is processed, the current 24 bit count along with the flow ID of the packet processed are written to static RAM. This data which is saved in SRAM can be read by an extern machine by sending special control cells. The default configuration is to write total values to successive SRAM memory locations, thereby allowing an external process validate the processing of each packet processed by TCP-Splitter and the ByteCount application. After performing 32 write operations to SRAM, the address is reset and future memory updates will overwrite previous entries.

The following setup operations are required in order to demonstrate the ByteCount circuit. First reset the switch and restart all ports using the NCHARGE web page. Load the ByteCount bit file on an available FPX card. Setup the FPX virtual circuit VPI:0 VCI:32 to route application data traffic from the switch to the RAD switch, from the RAD switch to the RAD line card, and from the RAD line card back to the switch. Define the FPX virtual circuit VPI:0 VCI:23 in the same manner to route memory read requests to the CCP module contained within the ByteCount circuit. Route VCI 50 traffic on the switch from a line card, to the FPX device running the ByteCount circuit, to a second line card. Using your favorite application(s), generate TCP/IP traffic that is sent through the switch. The ByteCount application is currently built with the *Defragment* circuit which only passes traffic sourced from port 8000 or destined for port 80. TCP/IP packets matching either of these filters will be counted by the ByteCount application. Current running total byte counts will be written to SRAM module 0 in successive memory locations. After sending some traffic through the switch, use the NCHARGE web site to read memory locations from SRAM 0 starting at byte zero. These memory locations should be filled in with flow identifiers and the current byte count.

6.2 TcpClient

The TcpClient application was written to showcase the operation of the *Defragment* circuit. The main purpose of the TcpClient application is to illuminate a LED on the FPX card while a HTTP web request (TCP port 80) is in progress. More specifically, when a start of flow is detected for TCP connection which passed the *Defragment* port filter, a LED is turned on. The LED continues to stay illuminated until a end of flow signal is processed for that same flow identifier. When surfing the Internet from a browser whose traffic is routed through an FPX card running the TcpClient application, a LED will blink on and off in unison with the active HTTP web requests.

Setting up the TcpClient application is similar to that of the ByteCount application. The major difference is that there is no need for a FPX virtual circuit for reading memory. After loading the TcpClient, add the FPX virtual circuit VPI:0 VCI:32 to route application traffic from the switch to the RAD switch, and from the RAD switch back to the switch. Set up the switch to route VCI 50 traffic throught the FPX module where the TcpClient application is running. When surfing the internet, you should see LED's 3 and 4 on the FPX card illuminate whenever a TCP connection is active.

6.3 Programmer

The Programmer application circuit is a rewrite of the Multi-Device Programmer circuit previously developed at this same facility [9]. This circuit listens for device programming information sent on TCP port 9000. This data is extracted from the TCP connection and sent to an adjoining FPX card.

An example of the operation of the Programmer application can be seen in figure 16. Data is routed into the Programmer circuit via the RAD-switch interface. Data is extracted from the TCP stream destined for port 9000. It is assumed that this stream data contains a series of 14 byte, AAL0 control cells. These extracted cells are sent to the RAD-line card interface to be routed by the NID to a target FPX device. An idle period of 150 clock cycles is inserted in between each control cell in order not to overrun the NID on the target device.

To test the Programmer circuit, mount a stacked pair of FPX cards in an available slot on the switch. Program the lower of the stacked FPX cards with the Programmer circuit. Set the FPX virtual circuit VPI:0 VCI:32 to route traffic from the switch to the RAD switch, and then back from the RAD switch to the switch. This VC is used for routing TCP data containing reprogramming information through the

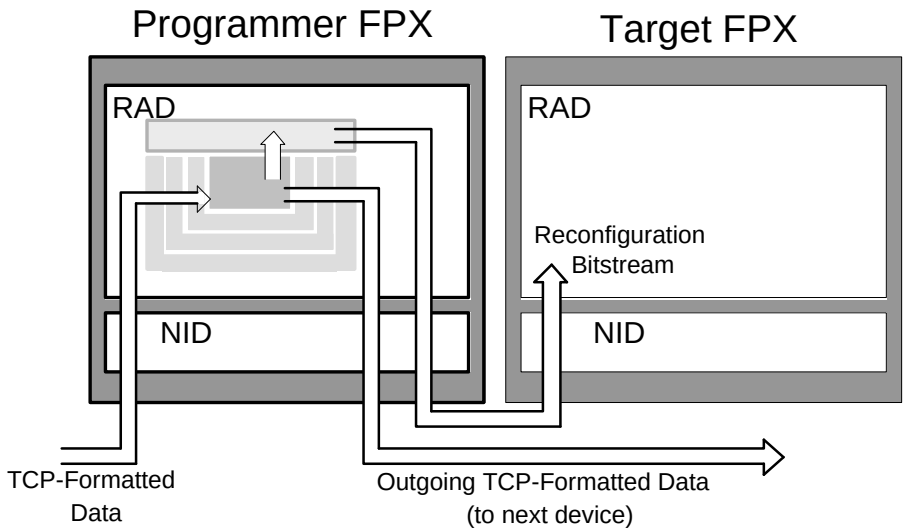

Figure 16: Programmer Circuit Operation

Programmer circuit. Set the FPX virtual circuit VPI:0 VCI:22 to route traffic from the RAD line card to the line card interface. This VC will contain programming information extracted from the TCP flow and will be routed to the upper of the two stacked FPX devices. A virtual circuit should be setup on the switch to route traffic from a line card to the stacked FPXs and then on to a second line card. Return traffic can be routed directly from line card to line card.

The Programmer application requires companion workstation applications which initiate and terminate the TCP connection over which the reprogramming information is sent. To fulfill this role, the utility programs *send* and *sink* were developed transmit a data file from one workstation to another.

The *sink* application is executed on a workstation connected to one side of the WUGS where the device programming is to take place. The TCP port number on which the *sink* program should listen is passed in as the only parameter. All data received by this application is discarded.

The *send* application is executed on a second workstation placed on the opposite side of the WUGS. In this manner, data sent between between the *send* and *sink* programs can be routed through the WUGS. Device programming information is read from the files `rot13client1.log` and `rot13client2.log`. These data files contain AAL0 control cells which contain commands and data necessary to program a FPX

device. The *send* program takes two parameters, the first being the IP address of the destination machine running the *sink* application, and the second being the TCP port number of the remote server.

The NID on the target device does not have the ability to flow control inbound control cells containing reprogramming information. For this reason, the programming information needs to be metered by the *send* application. With data metering in place, the current implementation of the Programmer circuit is able to program a FPX device in approximately 1.2 seconds.

7 Conclusion

The TCP-Splitter circuit supports processing of TCP flows in hardware. Client applications are provided with a simple interface from which they receive stream data. TCP packets are classified, checksummed and ordered so that the client application can concentrate on application level processing and not have to worry about protocol processing. The operation of the TCP-Splitter circuit, the *Defragment* companion circuit, and several client applications were described in this document.

References

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

[2] D. V. Schuehler and J. W. Lockwood, "Tcp-splitter: A tcp/ip flow monitor in reconfigurable hardware," in *Proceedings of Symposium on High Performance Interconnects (HotI'02)*, (Stanford, CA, USA), pp. 127–131, Aug. 2002.

[3] V. Jacobson and R. Braden, "RFC1072: TCP Extensions for Long-Delay Paths." http://www.faqs.org/rfcs/rfc1072.html, Oct 1988.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] H. Ku, J. W. Lockwood, and D. V. Schuehler, "TCP programmer for FPXs," Tech. Rep. WUCS-2002-29, Washington University in Saint Louis, Aug. 2002.