Report Number: WUCSE-2002-29

2002-08-27

TCP Programmer for FPXs

Harvey Ku, John W. Lockwood, and David V. Schuehler

Reconfigurable hardware platforms are the key to extensible high speed networks. They provide flexibility without hindering performance through the internet. Current development of the Field-programmable Port Extender (FPX), a reconfigurable hardware platform allows reconfiguration through an ATM network. However, majority of the internet today is based on the highly popular TCP/IP protocol. The contribution of this work will allow modular components to be reprogrammed via TCP/IP

Department of Computer Science & Engineering

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

2002-29

TCP Programmer for FPXs

Authors: Ku, Harvey; Lockwood, John W.; Schuehler, David S.

August 27, 2002

Abstract: Reconfigurable hardware platforms are the key to extensible high speed networks. They provide flexibility without hindering performance through the internet. Current development of the Field-programmable Port Extender (FPX), a reconfigurable hardware platform allows reconfiguration through an ATM network. However, majority of the internet today is based on the highly popular TCP/IP protocol. The contribution of this work will allow modular components to be reprogrammed via TCP/IP

TCP Programmer for FPXs

Harvey Ku
John W. Lockwood
David V. Schuehler

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

WUCS-2002-29
http://www.arl.wustl.edu/arl/projects/fpx

August 27, 2002

Abstract

Reconfigurable hardware platforms are the key to extensible high speed networks. They provide flexibility without hindering performance through the internet. Current development of the Field-programmable Port Extender (FPX), a reconfigurable hardware platform allows reconfiguration through an ATM network. However, majority of the internet today is based on the highly popular TCP/IP protocol. The contribution of this work will allow modular components to be reprogrammed via TCP/IP

1 INTRODUCTION

Reconfiguration of an FPX over a TCP/IP connection enables quick programming of one or more FPX hardware modules simultaneously over the Internet. Standard TCP client and server can be used to send a bit file to FPX devices along the path. The TCP server acts as a sink that collects and verifies transmission of reconfiguration packets on the receiving end. This implementation uses the existing protocol wrappers [1] and the TCP splitter [2] developed for the FPX [3]. As TCP flow containing the bit file is passed through the protocol wrappers, it is also passed through a TCP splitter that splits incoming TCP traffic into a client flow and an outgoing TCP flow that passes back out the wrappers. With this flow, traffic can be forwarded to targeted FPXs to reprogram a single or even multiple FPX hardware modules with a single transfer operation.

1.1 Overall Design Flow

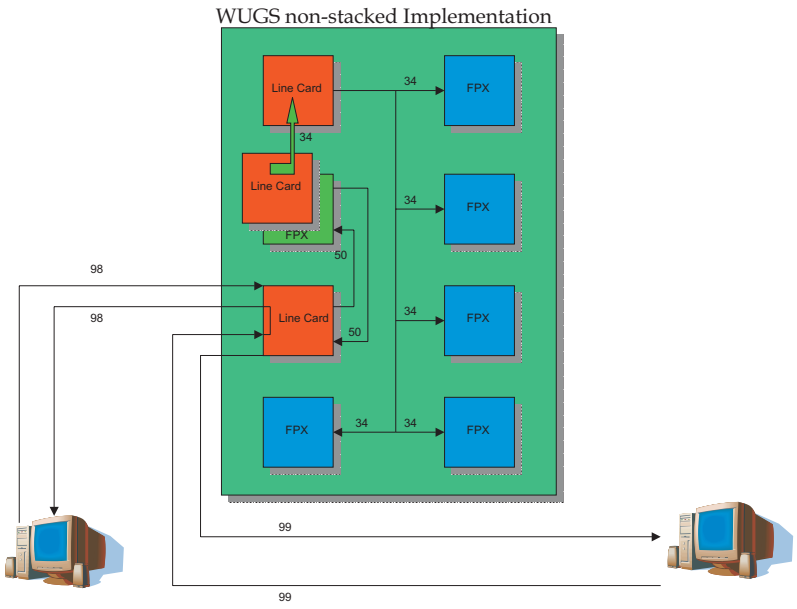

Figure 1: Non-stacked Implementation

Figure 1 is a design flow based on a non-stacked FPX implementation. A server is located on a remote machine that listens on a special port for an incoming reconfiguration request from a client. As the server

and client converses, a Washington University Gigabit switch (WUGS) monitors the connection between the client and server. The WUGS will then route traffic into an FPX module that has been initially loaded with the TCP Programmer, as shown in Figure 1 with the stacked linecard. Once the TCP programmer receives the TCP flow on VCI 50, it extracts control cells from the flow and generates a second flow on VCI 34, known as the client application flow. These control cells contain the bit file that will be used to reconfigure an FPX. The outgoing TCP flow is identical to the incoming TCP flow with the exception that the TTL field has been decremented by one and corresponding checksums altered.

The client flow is passed onto a stacked linecard which will forward traffic to a linecard on a different port. From that linecard, all traffic on VCI 34 can be either uni-casted or multi-casted to neighboring FPXs to be reconfigured. Traffic from the outgoing TCP flow will be sent back to the linecard it arrived from and routed to the destination Server. Once the server receives packets from the client, it will dump all incoming packets to a log file and send ACKs back to the client side so data flow will continue. ACKs are sent back through a slightly different circuit, it travels a direct route back to the client end without being passed through the TCP programmer. The flow will get redirected straight back out of the linecard.

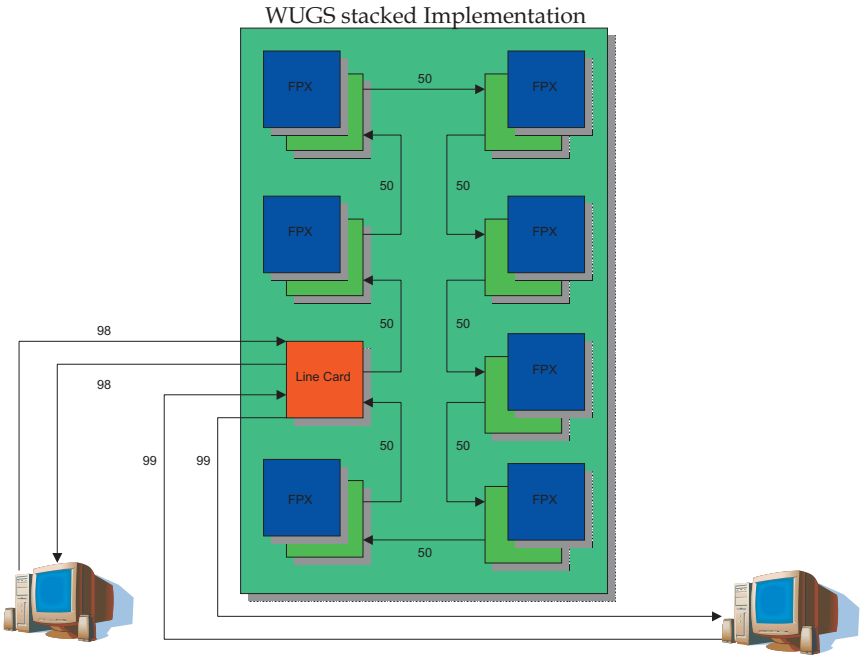

Figure 2: Stacked Implementation

Figure 2 shows a design flow based on a stacked FPX implementation. For every stacked FPX to be reconfigured, an initial FPX must be loaded with the TCP programmer. In this configuration, the client and server establishes a connection through a similiar manner as the non-stacked implementation. The WUGS acts as a monitor between the client and server, and sends incomming flow to a single TCP Programmer. However, instead of routing the client flow to a stacked linecard, it is routed to a stacked FPX to be reconfigured.

The outgoing TCP flow can be sent to either a neighboring FPX or back to the linecard port it arrived on. If it is sent to a second FPX, it will reconfigure that FPX and output a third flow that can be used to reconfigure a third FPX. This process can be repeated multiple times on the same switch, creating a daisy chain back to the initial linecard that incoming TCP flow entered on.

Once a stacked FPX is programmed, VCIs for the NID need to be initialized before incoming traffic can be routed to the stacked FPX. VCI control cells can be included along with the initial configuration.

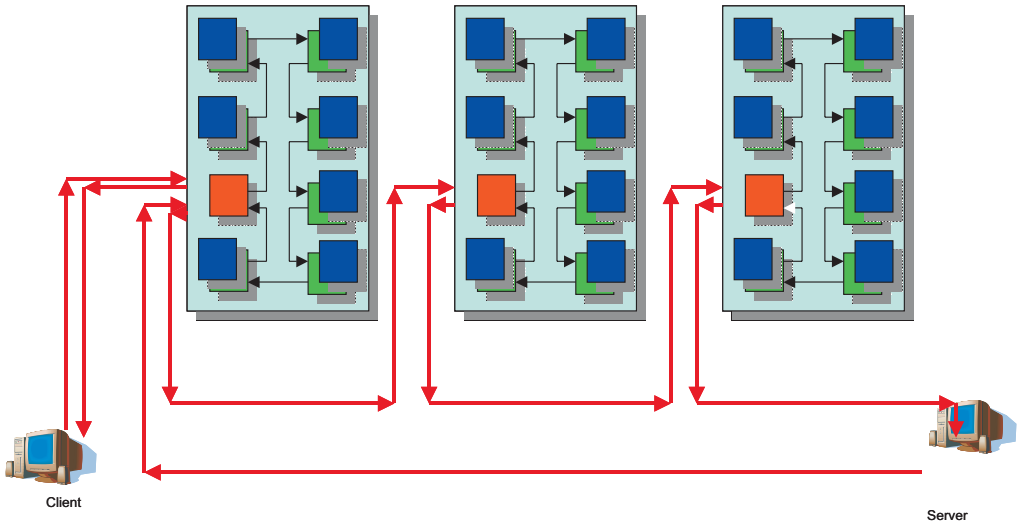

Figure 3: Multiple Switch Implementation

Multiple FPXs on separate switches may be programmed in the same manner with a single TCP file transfer as seen in Figure 3. The number of FPXs that a single TCP flow can reconfigure is virtually limitless, however it is still restricted by the TTL field on a TCP packet. As traffic is traversed from one FPX to another, the TTL field is decremented by one. If the number of FPXs ever exceed this number,

packets will start getting dropped by the Protocol wrappers and never reach the destination server.

1.2 Data Flow Architecture

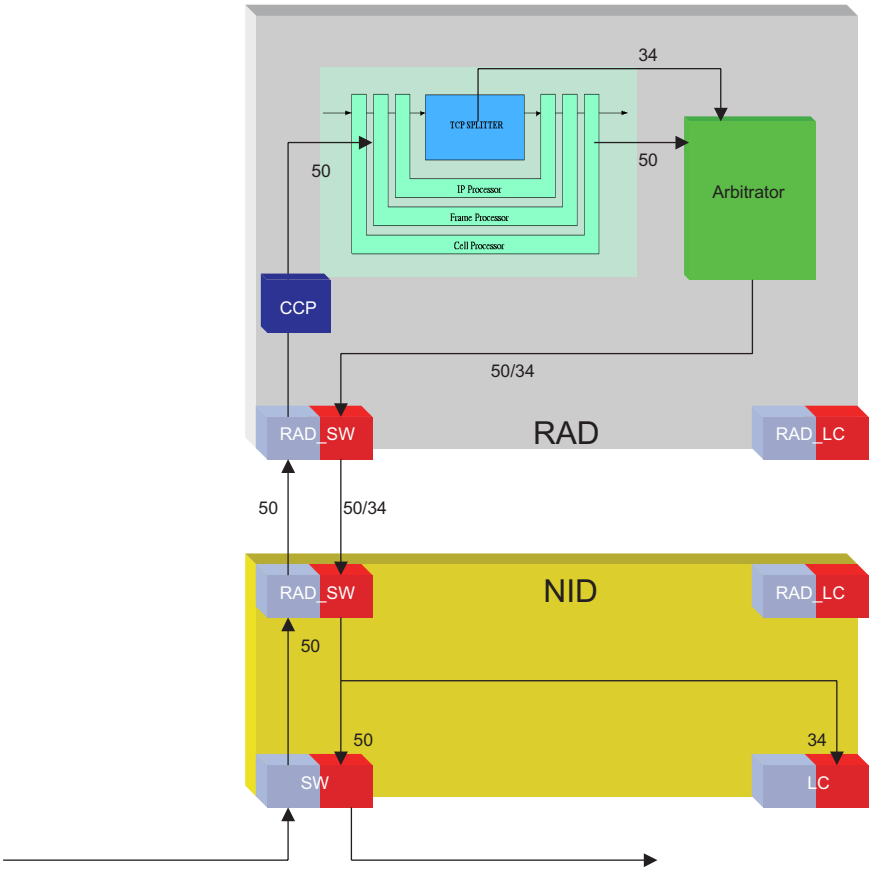

Figure 4: Data Flow within FPX

Figure 4 depicts data flow passing through a single FPX. As TCP traffic is generated from a client, it is casted to a VCI of 50 (decimal) before it enters the NID on a specific FPX. The reason for this is that the Protocol wrappers process packets only on VCI 50. As incoming traffic enters the NID on the SWITCH port, it will only forward all flows on VCI 50 to the RAD_SWITCH port to be processed by the RAD. TCP traffic in the RAD is first passed straight through a control cell processor that does not modify any cells. This current implementation does not make use of the CCP.

Following the CCP, flow is passed on to our Protocol wrappers to process our incoming TCP packets. First it is passed through a Cell processor which will process raw ATM cells, a Frame Processor to process variable length AAL5 frames, a IP Processor to process IP packets, and finally through a TCP splitter which will generate our client flow and output the outgoing TCP flow. The outgoing TCP flow will remain on VCI 50, while client flows are generated on VCI 34 (decimal).

Both traffic flows will enter an Arbitrator module that will buffer each incoming flow and determine the order of the flows being clocked out. Flows will alternate turns to be clocked out into the RAD_SWITCH port. However, the outgoing TCP flow is given priority to be clocked out over the client flow. As flows enter the NID, it will separate these mixed flows and route them to different outgoing ports. Flows on VCI 50 will be forwarded to the SWITCH port, while flows on VCI 34 will be sent to the lincard port.

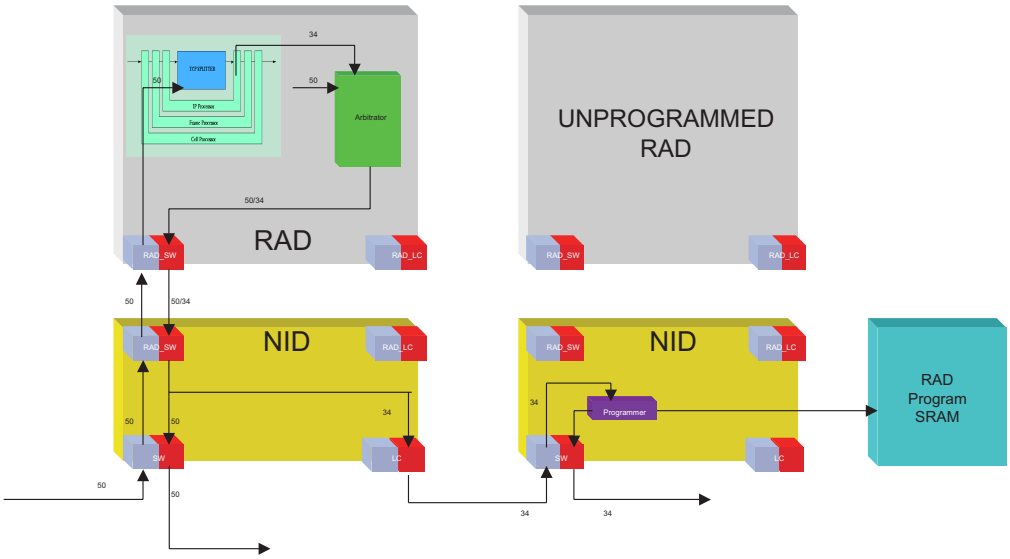

Figure 5: Data Flow for a stacked implementation

As the client flow exits the linecard port, it may enter either a stacked lincard or a stacked FPX. In the case of a stacked FPX, flows on VCI 34 will enter the NID to be intercepted by the NID programmer. VCI 34 is a special VCI for the NID which recognizes it as a control cell that contains reconfiguration data. The NID programmer will initiate writes to the RAD Program SRAM, during the reconfiguration process. However, there needs to be two separate loads into SRAM because a bit file is too large to be completed in a single load. The size of a bit file is 1,241 KB, therefore it needs to be partitioned as 782KB for the first load

and 458 KB for the second load. There must also be a wait of one second between the two loads in order to safely read out the first file from SRAM before loading the second file.

2 INTERFACES

2.1 Arbitrator

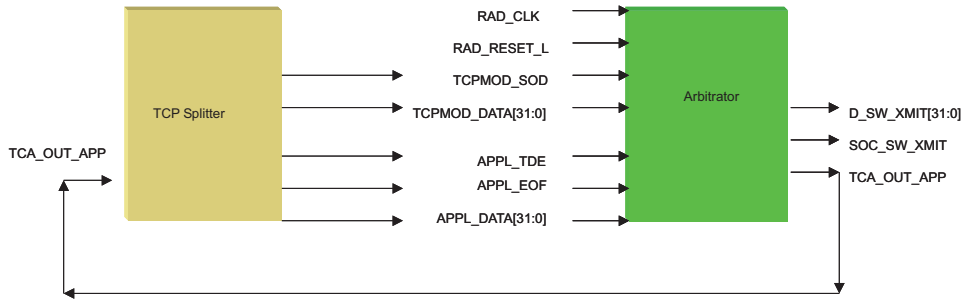

Figure 6: Interface between TCP Splitter

Figure 6 shows the interface between the Arbitrator and the TCP Splitter. The outgoing TCP flow corresponds to the TCPMOD_DATA[31:0] and TCPMOD_SOD signals, while the client flow corresponds to the APPL_DATA[31:0], APPL_TDE and APPL_EOF signals. Congestion control for the Client Application is handled by the TCA_OUT_APP signal which de-asserts the signal and pushes it back upstream to the TCP Splitter to notifty the TCP Splitter that congestion is occurring. The TCP Splitter will push this signal back upstream to the NID to halt traffic until TCA_OUT_APP is reasserted.

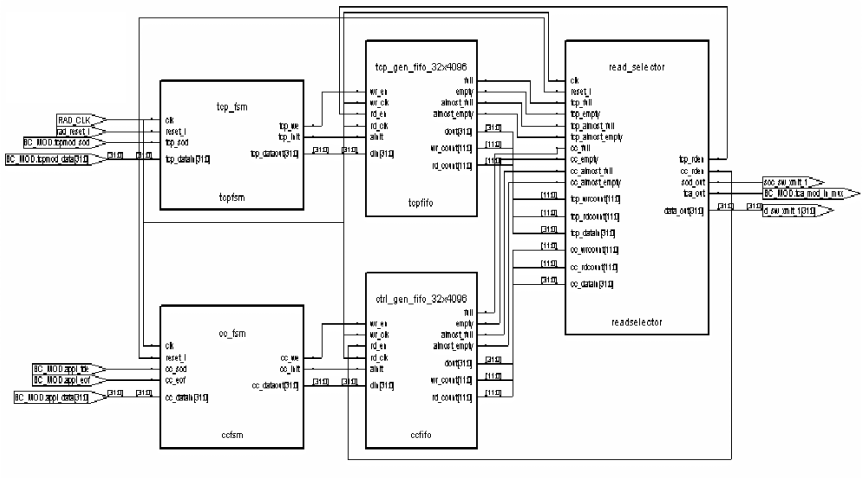

Figure 7: Arbitrator

As seen in Figure 7, the Arbitrator contains five main components, a TCP_FSM, TCP_GEN_FIFO_32X4096, a CC_FSM, a CTRL_GEN_FIFO_32X4096 and a READ_SELECTOR. The TCP_FSM is a fifo controller that enables writes into the TCP_GEN_FIFO_32X4096 fifo. The CC_FSM is also a fifo controller that enables writes for the CTRL_GEN_FIFO_32X4096. The sizes of these two fifos are 32 bits by 4096, which is sufficient for this application. However, sizes could be increased to reduce the frequency that a TCA signal is deasserted. The READ_SELECTOR, connects the two fifos and determines the next fifo to be clocked out.

2.2 TCP_FSM

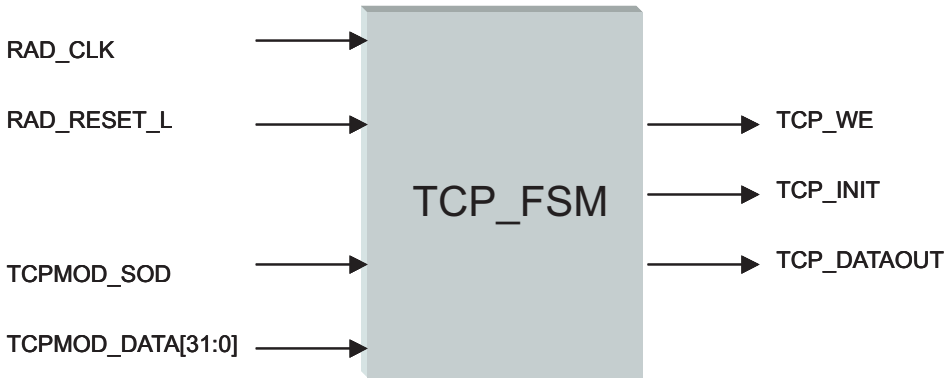

Figure 8: TCP FSM struct

In Figure 8, the TCP_FSM requires only two input signals, TCPMOD_SOD and TCPMOD_DATA[31:0]. TCPMOD_SOD represents the start of cell bit that signals when a cell is ready to be written into the fifo. TCPMOD_DATA represents the data input to be written into the fifo. The output signals of the system are TCP_WE, TCP_INIT and TCP_DATAOUT. Upon reset, TCP_INIT is asserted to initialize the fifo to be used. TCP_WE is a signal that will notify TCP_GEN_FIFO to enable writes onto the fifo.

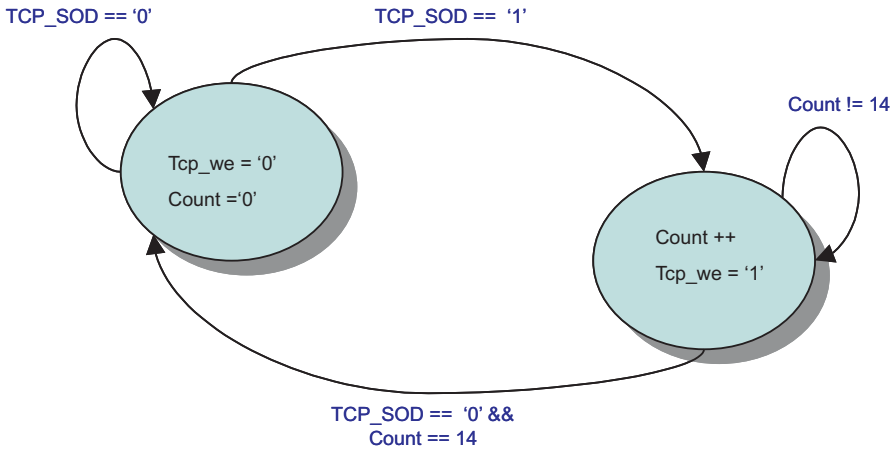

Figure 9: InternalTCP FSM behavior

The behavior of this FSM can be seen in Figure 9, it contains two states, a START state and an ACTIVE state. The machine begins at the START state where counters and output bits are initialized. Upon receiving a TCP_SOD = = '1', it will make a transition to the ACTIVE state where the TCP_WE bit is set and data begins to be written to the external FIFO. During the ACTIVE state, a count signal is incremented by one on every word that is clocked out. The moment that count reaches 14, it will reset itself back to 0. If TCP_SOD is also 0 at this time, it will make a transition back to the START state waiting for another TCP_SOD signal. If TCP_SOD is still high when count reaches 14, then it will remain in the ACTIVE state till there are no more TCP_SOD assertions.

2.3 CC_FSM

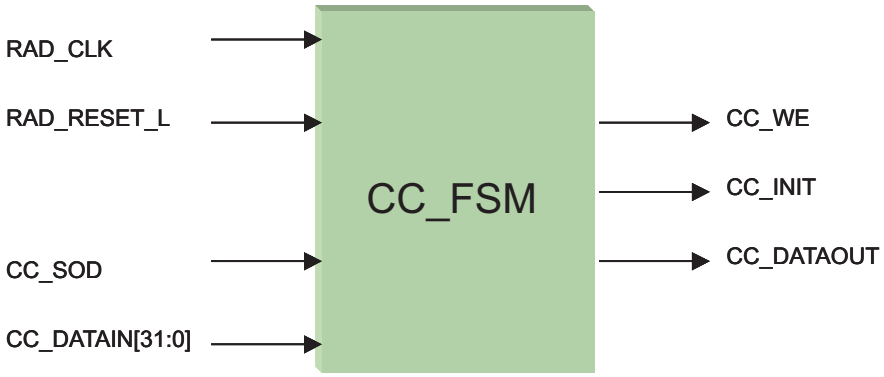

Figure 10: CC FSM struct

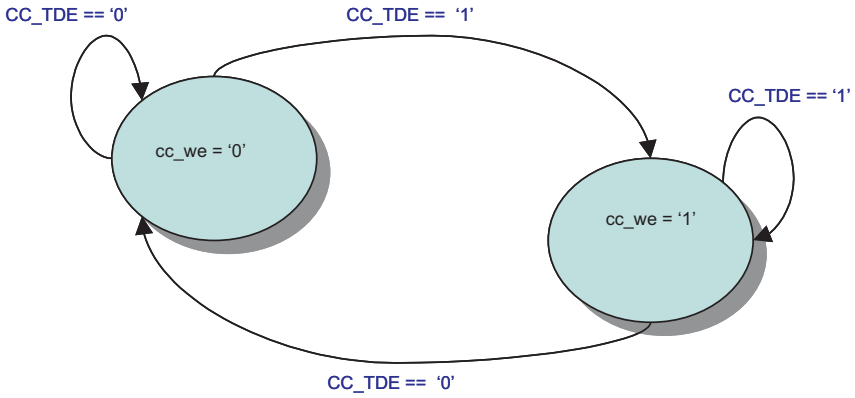

Figure 11: Internal CC FSM behavior

The internal behavior of the CC_FSM in Figure 11 is very similar to that of the TCP_FSM with one main exception. Instead of having a start of cell signal that asserts to high for one clock cycle to notify a start of cell, we have an enable bit set to clock data out to the external fifo. The client application does not send out any notification signaling a start of cell, it will only send a signal called CC_TDE. The CC_TDE is a data enable bit that asserts to high while data words being output are valid. With this enable bit, we are no longer required to have count 14 words for an entire cell. We are able to clock in any data while the CC_TDE bit is

valid. There are also two states in this fsm, a START state and a ACTIVE state. In the ACTIVE state, the system will continue clocking data into the FIFO as long as CC_TDE is asserted high.

2.4 Read Selector

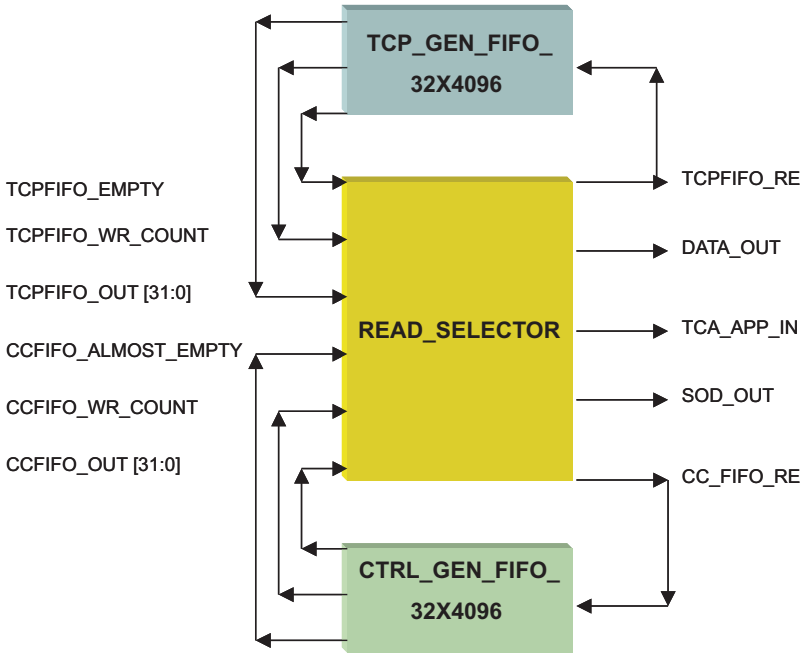

Figure 12: Read Selector Structure

The READ_SELECTOR controls reads from the two fifos with the TCPFIFO_RE and the CC_FIFO_RE signals. The two fifos will then pass three other signals back to the READ_SELECTOR as shown in Figure 12, one to notify how full the FIFOs are, one whether or not it is empty and the 32 bit data it has read. After obtaining cells from one of the two FIFOs, it will assert a SOD_OUT signal and output to DATA_OUT.

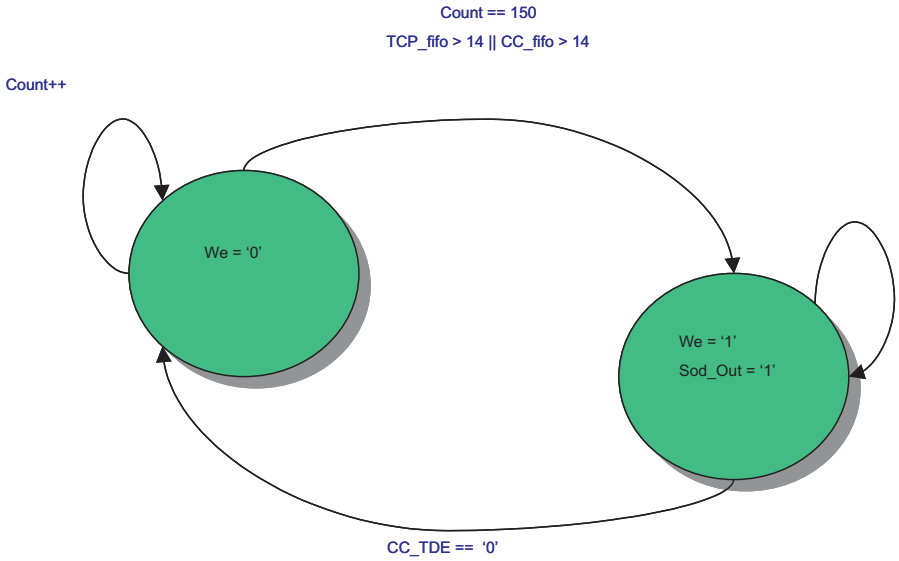

Figure 13: Internal Read Selector FSM behavior

In Figure 12, the state machine functions with two states, a START and an ACTIVE state. Upon reset, the system will begin in the START state, while in the START state, it can either decide to remain in the START state or transition to the ACTIVE and set off a flag bit. If a transition is made to the ACTIVE state, the flag bit must be set to one or zero, a zero indicates that we wish to read from the TCP_GEN_FIFO and a one indicates a CC_GEN_FIFO read. The decision to set this flag bit is based on the priority level that we wish to output the two flows. The current implementation chooses to always select the TCP flow as long as there is a complete cell available in that buffer. A complete cell contains 14 words, once the TCP_GEN_FIFO becomes empty, we are able to start outputting from the CC_GEN_FIFO. The reason for selecting the TCP_GEN_FIFO over the CC_GEN_FIFO is because we do not want to delay the TCP output flow from reaching the next FPX it wishes to reconfigure. The server must also constantly communicate with the client in a timely efficient manner.

If a transition is made to the ACTIVE state, we initialize a count to zero and begin reading in from one of the two fifos for data, while at the same time asserting a SOD_OUT for one clock cycle to signal a start of cell. Once count reaches 14, we have clocked out an entire cell and we are able to transition back to the START state. To support the rate at which the NID is able to accept back to back cells, we must insert

a delay between each cell we clock out. To accomplish this, once a single cell has been clocked out, we return back to the START state and sleep in this state for a specified number of clock ticks. This will prevent continous cells from overflowing the NID's buffer. The number of clock ticks can be modified by changing the start_count signal.

Besides overflowing the NID's buffer, it is also possible to overflow FIFO's on the RAD. To be sure that they do not overflow, a TCA signal must be deasserted when congestion occurs. The READ_SELECTOR handles this congestion by deassertting a TCA_OUT signal when the CC_GEN_FIFO_32X4096 reaches 3000 words. Upon reaching this limit, data flow into the Arbitrator will cease and the TCP_GEN_FIFO will soon empty itself. As soon as the TCP_GEN_FIFO empties, the CC_GEN_FIFO will then be able to start clocking out data and not overflow.

3 SIGNALS

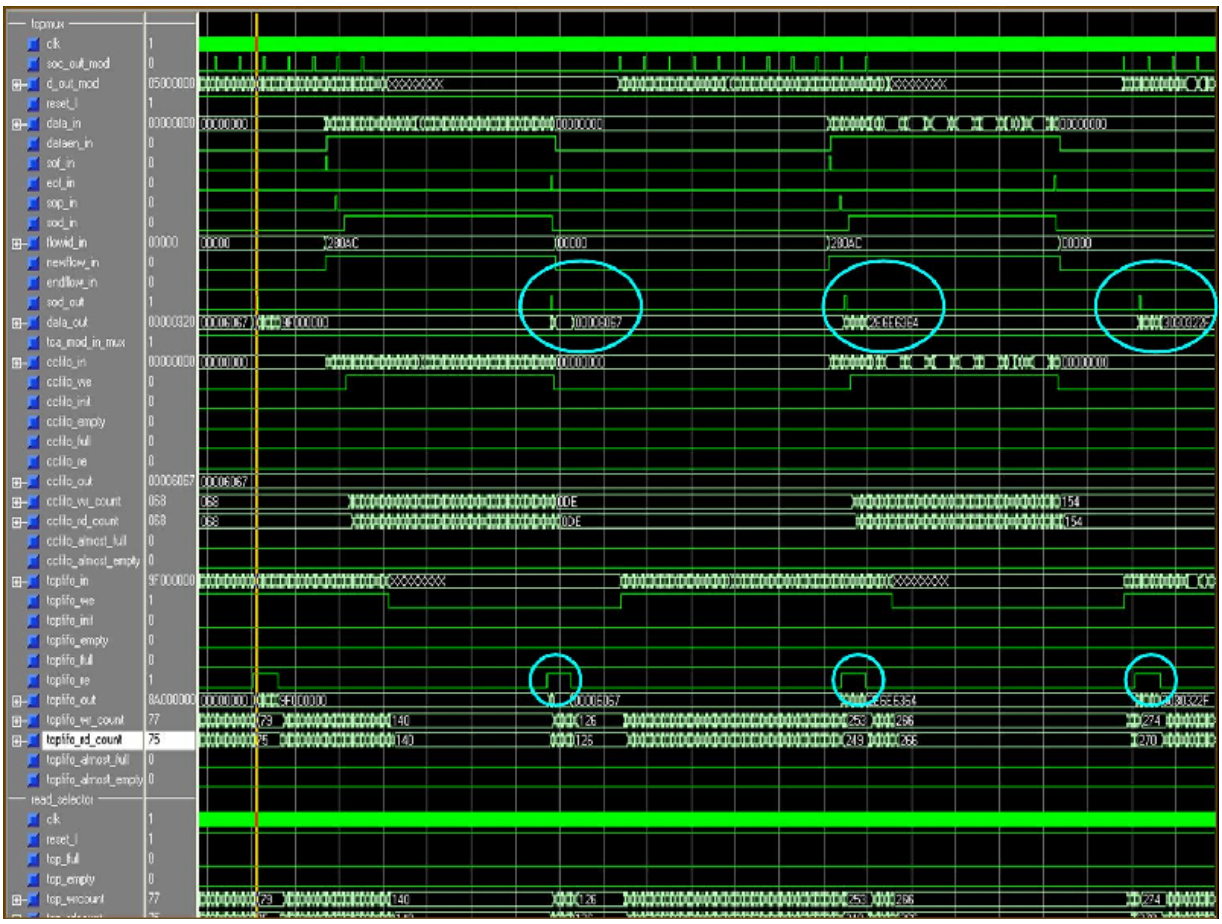

Figure 14: wave forms

Figure 14 shows output signals from the Arbitrator, from the top row you can see that a start of cell signal is raised for one clock cycle with data_out following right below. This implementation shows a 150 clock cycle delay between each cell being output in this wave diagram. The bottom signals are read enable signals that read from the fifos.

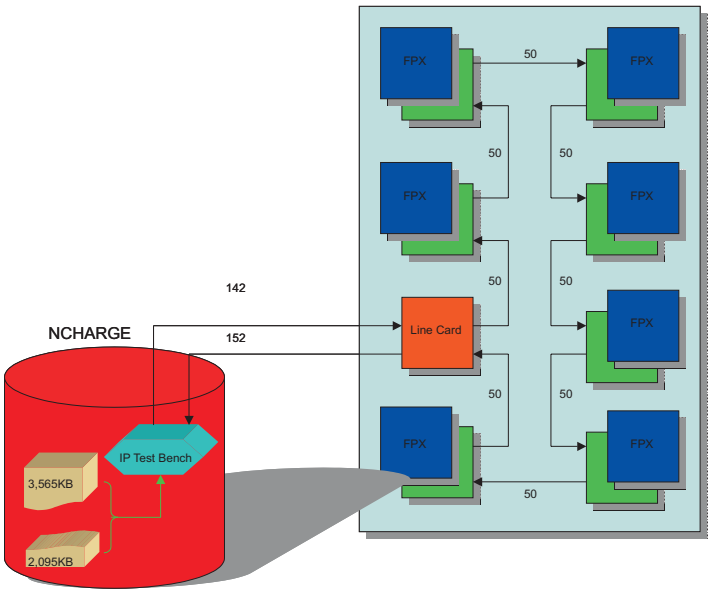

Figure 15: Testing with NCHARGE

4 TESTING WITH NCHARGE

Testing with an actual TCP Client/Server Application at first may present a lot of complications due to variable TCP frame sizes and data rates. In order to avoid these problems during initial testing, we can use NCHARGE and IPTestbench to simulate a TCP/IP data flow. The following are steps used to program two stacked FPXs on port zero and port one, with traffic incoming on a linecard on port three.

4.1 NCHARGE Steps

I. Convert bit file into Control Cell format

a. Go into /usr/tmp/TCPprogrammer directory on fpx2.arl.wust.edu and run the following script TCPcontrolformat.sh

i. Rename your bit file to input.bit

ii. Place bit file into /formatbitfile directory

iii. Run ./formatcells.sh, your output file is output.txt(2,800KB) and output2.txt(1,645KB)

II. Set up VCIs to begin test

a. Program ports one and two with TCP Programmer

b. Set up the following VCIs

i. Unicast(3, 50, 2, 50) Unicast (2, 50, 1, 50) Unicast (1, 50, 3, 153)

ii. This can be done through NCHARGE's webpage or by running a Jammer Script on the local FPX machine

1. To run Jammer script go into the /project/arl/fpx/NCHARGE/cgi-bin directory and type in ./fpxswitch hku.js
2. hku.js is a default script that I modify for my testings, you can copy that and edit your changes
3. This is very helpful for debugging purposes, it allows multicasts
    a. You can do a multicast to an unused port and sniff traffic
    b. To listen on a port run the command

./dump_vcs 1 "VCI" > outputfile.txt

c. Set up VCI's on the NID on both FPXs

i. Use NCHARGE's webpage and route VCI 32(hex) from (SWITCH to RAD_SW) and back out from (RAD_SW to SWITCH).

Route VCI 22(hex) from(RAD_SW to LINECARD), from (LINCARD to RAD_LC) and (RAD_LC to RAD_LC)

III. Start IP Test bench on NCHARGE webpage and send OutputA.log to port 2 on VCI 142

a. Send output2.txt after output.txt completes

IV. Ports one and two should be programmed at this point

V. To facilitate going through the web process every time, use the snd_gen_cells command

a. ./snd_gen_cells 142 /usr/tmp/temp_ip2raw.txt > /dev/null

b. 142 is the sending VCI, /usr/tmp/temp_ip2raw.txt is the file IP Test Bench generated

c. /dev/null is the outputfile that comes back on VCI 152

5 TESTING WITH A REAL TCP CLIENT/SERVER APP

Programming a stacked FPX over a real TCP connection is similar to the steps in testing with NCHARGE, with one additional step during the file conversion process. After running ./TCPcontrolformat.sh and obtaining output files A and B, we will need to run it through a hex editing program in order to convert the ascii representation of hex characters into a byte stream file. The use of Hex Workshop is recommended for this process. After obtaining the two byte stream files, be sure to rename them to inputA.bit and inputB.bit and place the files under the directory /usr/tmp/hku/tcpclient/. Then run the command ./tcpserver 8765 on one machine and ./tcpclient 192.168.90.1 8765 on separate host. The number 8765 is the special port number that the server listens on and 192.168.90.1 is the destination IP address the client wishes to make a connection to. By running the tcpclient, it will automatically send one file right after another to program an FPX.

6 CELL FORMATS

6.1 Control cell transformation

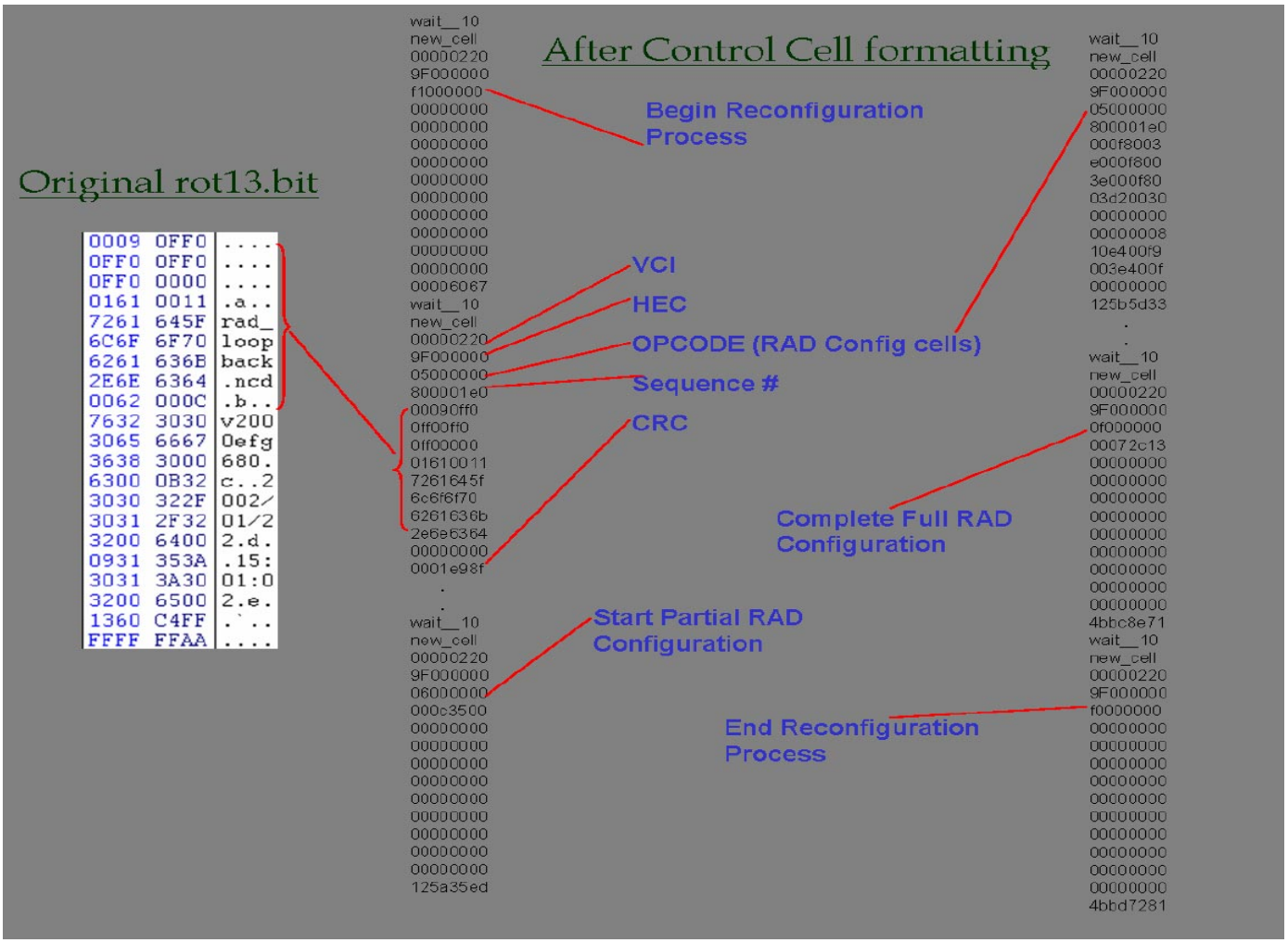

Figure 16: Control cell transition

Figure 16 depicts the required transitions that a bit file needs to be altered in order to reconfigure an FPX over TCP. Starting from the left side, an original rot13 bit file is passed through TCPcontrolformat.sh and becomes formatted into control cells. Note that all these cells become AAL0 frames with a VCI of 0x22. To determine how the file should be split up, do a search for the opcode 0x06000000, this signals

the NID to start programming the RAD with what is loaded in SRAM currently. The second half of the file should start with an opcode of 0x05000000 and end with an opcode of 0xf0000000 signaling the end of the reconfiguration process. However, by running the shell script, it will automatically partition the bit file into to two separate log files and generate a byte stream.

6.2 TCP packet transformation

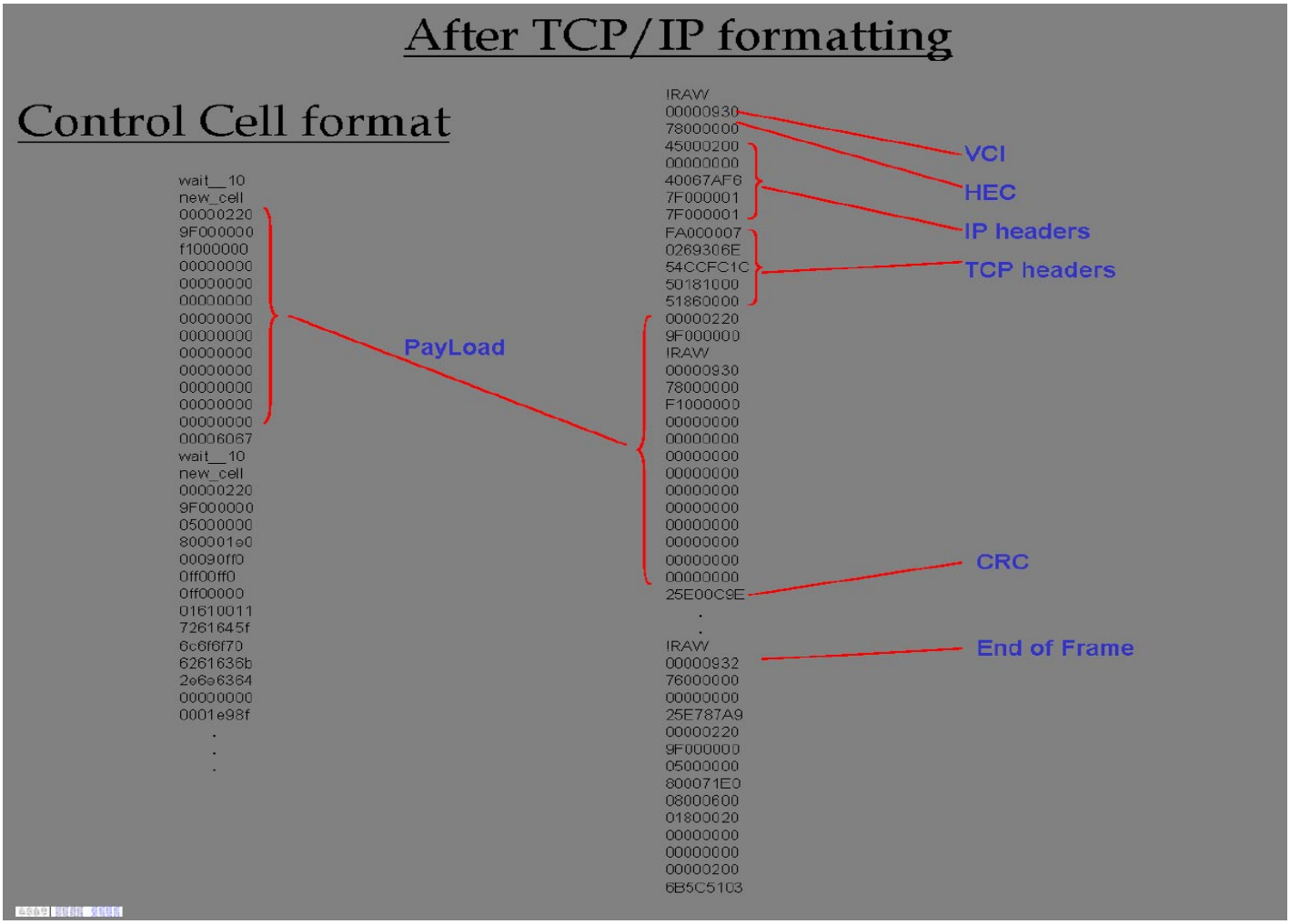

Figure 17: TCP packet transition

After aquiring cells in control cell format, the TCP Client Application or Ncharge's IP Test bench will format these control cells into TCP packets over AAL5 frames. Figure 17 shows various ATM, IP and TCP header fields as well as the payload on a single TCP frame.

7 RESULTS

The TCP Programmer has been fully implemented in VHDL and Synthesized. The design runs at a maximum frequency of 71.777 MHz. The TCP Programmer, TCP Splitter and IP protocol wrappers utilizes 52% of BLOCKRAMs and 27% of SLICEs on a XILINX XCV2000E chip. A TCP client and server application has been written to simulate a TCP reconfiguration of an FPX on a WUGS switch. The entire design has been tested and used to program multiple FPXs, multiple stacked FPXs, and multiple stacked FPXs on multiple switches. The design has not been fully optimized at the moment, each configuration takes approximately 55 seconds to fully send a bit file to a single or multiple FPXs. This requirement does not represent the time for each FPX to be programmed, multiple FPXs may be programmed at the same instance with this 55 seconds requirement.

8 FUTURE WORK

Although the TCP programmer is able to program multiple stacked FPXs in various configurations, there has been a few noted instances of corrupted data being passed the NID under a stacked configuration. During these occurrences, the NID would lock up and cease data flow into the TCP Programmer requiring a hardware restart. The exact cause of this glitch has not yet been determined, however power distribution under a stacked configuration seems to contribute to this likelihood. This glitch has never been witnessed under a non-stacked configuration. Future work may include determining the exact cause of this glitch.

Another improvement for the TCP Programmer is to optimize the rate of transferring a bit file across the network. Currently, the TCP/Client Application sends data at a rate of 400 bytes per 25 microseconds. This rate has been tested to be the most stable condition for the application to work properly. Higher rates cause the RAD to deassert a TCA signal that never gets reasserted. Further testing of the TCA signal can be done on the RAD and NID to determine where the TCA reassertion signal is lost. File transfer rates could potentially reach 5 seconds for a complete transfer.

With the current design of the TCP Programmer and the NID, it is not possible to export the design on the NID. However, with new modifications to the NID, Protocol wrappers and the TCP Programmer, we may be able to accomplish this task one day. This will free up the RAD on an FPX to allow more reconfigurations on fewer FPX cards. However, the TCP Programmer must listen on a special VCI that corresponds to the Programmer flow to allow processing on the special VCI.

9 ACKNOWLEDGEMENTS

Special thanks to John Lockwood and David V. S Schuehler for their coordination of the TCP Programmer. Also, thanks to Todd Sproull for his assistance in testing the application with NCharge [4]. This project would not have been possible without their guidance and support.

References

[1] 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.

[2] J. W. Lockwood and D. V. Schuehler, "Tcp-splitter: A tcp/ip flow monitor in reconfigurable hardware," in *Proceedings of Symposium on High Performance Interconnects (HotI'02)*, (Stanford, CA, USA), pp. 1–5, Aug. 2002.

[3] J. W. Lockwood, "Evolvable internet hardware platforms," in *The Third NASA/DoD Workshop on Evolvable Hardware (EH'2001)*, pp. 271–279, July 2001.

[4] 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.