Gigabit Ethernet TCP Regular Expression Matcher

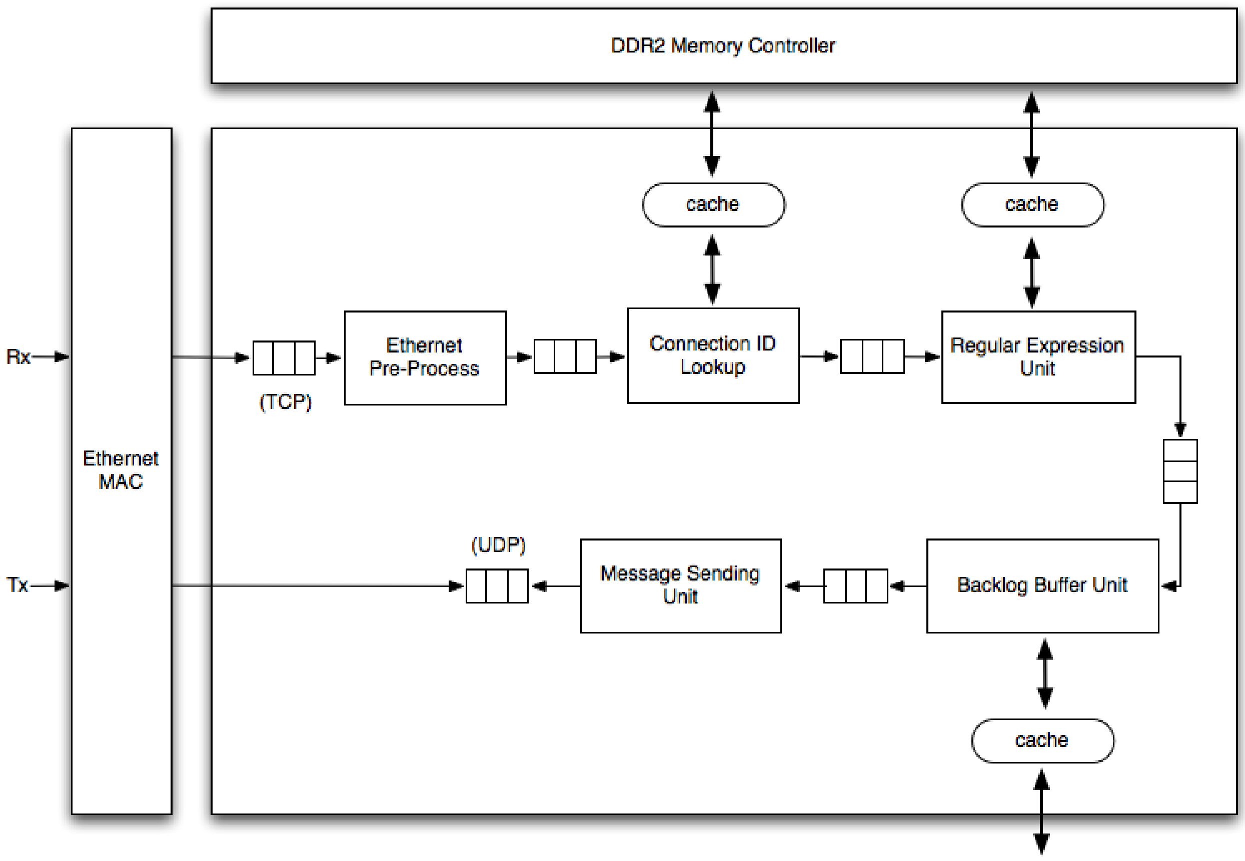
Ben Gelb, Sam Gross, Adam Lerer
6.375 Spring 2009

Introduction

In this paper, we present an FPGA implementation of a system that passively monitors an Ethernet network and performs stateful regular expression matching on TCP streams passing across the network. All future traffic from matching streams are forwarded via UDP to a monitoring computer, as well as 2KB of 'backlog' from before the point of the match. The system supports up to 64 regular expressions across up to 65,536 simultaneous TCP connections. We achieve successful processing of Ethernet traffic at up to 800Mbps, running at 100MHz on a Xilinx Virtex-5 FPGA.

Micro-architecture Overview

Our system processes a stream of incoming Ethernet packets by passing them through a variety of modules. First, frames are decoded off of the wire with an off-chip PHY which presents data to the pins of the FPGA using a standard GMII interface. This data is then routed to an on-chip hard-silicon MAC provided on our chosen FPGA (the Xilinx XC5VLX110T). A Xilinx HDL wrapper presents a simple FIFO-style interface (called LocalLink) for both the transmit and receive paths from the MAC.



Next, received Ethernet frames are pulled from the LocalLink interface into the core of our processor. First, they pass through the Ethernet Pre-processor module, which parses the Ethernet, IP, and TCP headers to discover the source and destination IP addresses and port numbers of an incoming packet. Additionally, any non-TCP or malformed packets are discarded at this step.

Next, the payload of TCP packets, along with select connection information including the IP and Port numbers, as well as TCP flags, are passed to the Connection Lookup module. This module maintains a 256K element hash table of TCP connections and mappings to an internal 16-bit "connection ID" which is associated with a particular connection. If a connection is not found in the hash table, a new entry is created, and a new connection ID is allocated from a free list of 64K possible identifiers. This limits the total number of connections that may be tracked simultaneously to 64K. The output of the Connection Lookup module is sent to the Regular Expression Matcher.

The Reguar Expression Matcher applies a collection of regular expressions to the data stream. Each regular expression is described as a discrete finite automaton (DFA) which maps to an FSM in the FPGA. The collection of FSMs are all run in parallel on the datastream. At the end of a packet, the state of the FSMs is saved away into main memory. At the beginning of each packet, the connection ID from the Connection Lookup module is used to load the appropriate state into the FSMs. This allows patterns to be matched on TCP streams - across individual TCP packets and across multiple connections. In the event a regular expression match occurs, a "match" message is inserted into the datastream leaving the Regular Expression Matcher.

The output of the Regular Expression Matcher is fed to the Backlog Buffer Unit. This module keeps a log of the most recent 2 kilobytes in a stream. This is done by maintaining a ring buffer for each connection in memory. If a "match" message is received, the entire contents of the backlog buffer for that particular connection, as well as any future data from that connection, is forwarded to the UDP Sender module.

The UDP Sender accepts data from the Backlog Buffer Unit and generates Ethernet frames containing statically-addressed UDP packets for transmission out of the Ethernet interface via the Ethernet MAC and PHY hardware. The UDP Sender generates two types of packets - match packets, and data packets. Both types of packets contain a basic header, consisting of the source and destination IP and port numbers of the matched connection as well as the internal connection ID of the matched stream. Additionally, match packets contain the index of the regular expression that caused a match. Data packets contain a length field, followed by length bytes of data.

Memory Subsystem
Several modules in our system use amounts of memory that necessitate an off-chip RAM.

- Connection Hash Table - 128 bits * 256K = 4MB
- Connection ID Free List - 16 bits * 64K = 128KB
- Regular Expression Matcher State - 8 bits * 64 regular expressions * 64K = 4MB
- Backlog Buffer Unit - 2KB * 64K = 128MB

Accordingly, we used a 256MB DDR2 SDRAM for our main memory, along with several layers of logic to present a consistent, usable interface to the core logic. The DDR2 SODIMM interfaces with the Xilinx Multi-port Memory Controller (MPMC) IP core. This core provides up to 8 memory access ports with a configurable "personality" that defines the interface of the port. For interfacing to our logic, we chose to use the Xilinx Native Port Interface (NPI) which uses a set of 3 FIFOs - read, write, and command.

A Bluespec interface called NPIWires was created that defined always_ready, always_enabled methods for each of the wires in the NPI inteface. A second interface was created that provided Get and Put methods for each of the NPI FIFOs. A NPIMaster module was created that implemented both of these interfaces and a some simple rules to tie them together. This module also includes a clock-domain crossing, so that the MPMC clock and the core logic clock can be run at different speeds.

In order to further abstract away the particular memory implementation, a simple blocking writeback cache module was created. This module connects to the NPIMaster module to access memory, and provides a standard Server interface to service single-word read and write requests. The word size was made a parameter of the cache so that the cache could be easily used in each of the four required places. The caches communicate over the NPI interface by performing burst reads and writes of 16 64-bit words, regardless of the desired data port size.

In our design, four caches are used for each of the four items listed above. Each cache has its own corresponding NPIMaster modules, and NPIWires interface exported out of the core logic. Each NPIWires interface connects to a different port on the MPMC, which takes care of arbitration. Note that there are no scheduling concerns related to shared memory acces, because although all of the modules utilize the same physical memory chip, each uses an independent area of memory.

Core Logic - Tagged Union Types
Two tagged unions were created to facilitate communication among our modules. First, the RawPacket type was created for use between the Ethernet Pre-Processor and the Connection Lookup module. It simply captures the important aspects of a TCP packet.

```
typedef union tagged {
    IPAddr RawSrcIP;
    IPAddr RawDstIP;
    struct {IPPort srcPort; IPPort dstPort;} RawPorts;
    TCPSeqNo RawSeqno;
    TCPFlags RawFlags;
    PayloadLength RawPlen;
    Payload RawP;
} RawPacket deriving(Eq, Bits);
```

The SystemPacket type is used in the rest of the system for inter-module communication. It is similar, but includes two "command" elements - NewConnection and ChangeConnection, that are used to forward around the internal connection ID assigned to a connection. NewConnection indicates a newly-allocated ID, and causes any modules with state to clear their state as it passes through the system. The ChangeConnection element indicates the start of a new packet on a different connection ID, and induces modules to load the correct connection-specific state.

```
typedef union tagged {
  ConnIndex NewConnection;
  ConnIndex ChangeConnection;
  IPAddr SrcIP;
  IPAddr DstIP;
  PortPair Ports;
  Payload ConnPayload;
  RegexIndex Match;
} SystemPacket deriving (Eq,Bits);
```

Core Logic - Ethernet Pre-Processor
The Ethernet Pre-Processor reads raw Ethernet frames in from the LocalLink interface. It uses a simple FSM to parse the incoming headers. It makes the following checks

- Packet is an IP packet
- Packet is an IPv4 packet
- Packet is a TCP packet

If any of these criteria fails, the packet is dropped. If all three criteria are met, a RawPacket type is generated, and enqueued in an outbound FIFO for transmission to the Connection Lookup module.

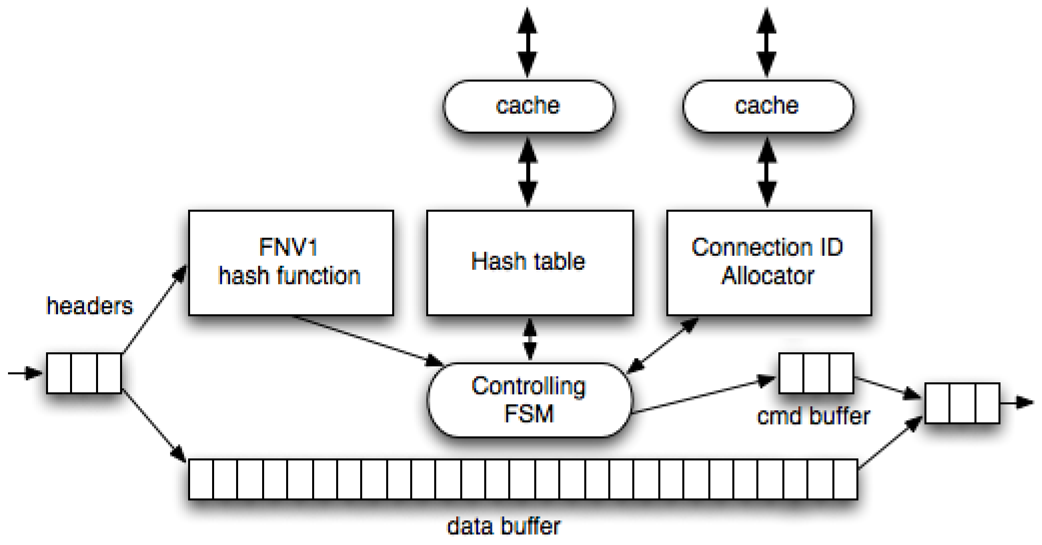
Additionally, packets that are less than 60 bytes are padded with 0s trailing the Ethernet payload prior to being transmitted. This is because Ethernet specifies that Ethernet frames must be at least 60 bytes long. If padding is added, it is stripped off in the Ethernet Pre-Processor prior to being forwarded.

Core Logic - Connection Lookup Modules
The connection lookup module is responsible for mapping TCP connections to the internal connection ID used within the system. We use the two IP address and port numbers to uniquely identify connections. A hash table contains the mappings from TCP connection to connection ID. The connection ID allocator is responsible for keeping track of free connection IDs.

We use the FNV1 hash function to hash the 96 bits of addresses and ports to an 18-bit index to the hash table. The FNV1 function requires only a single multiply and XOR for each hashed byte, while still providing good enough spreading for our needs.

The hash table contains 2^{18} slots, of which at most 2^{16} contain valid entries. Ensuring that the hash table is sparsely populated decreases the expected number of collisions at only the small cost of extra memory. We use linear probing to resolve collisions, since it can take advantage of the DDR2 burst reads. We can handle up to three collisions before suffering the latency of another roundtrip access to memory.

The connection ID allocator uses a ring buffer in memory to keep track of available connection IDs.



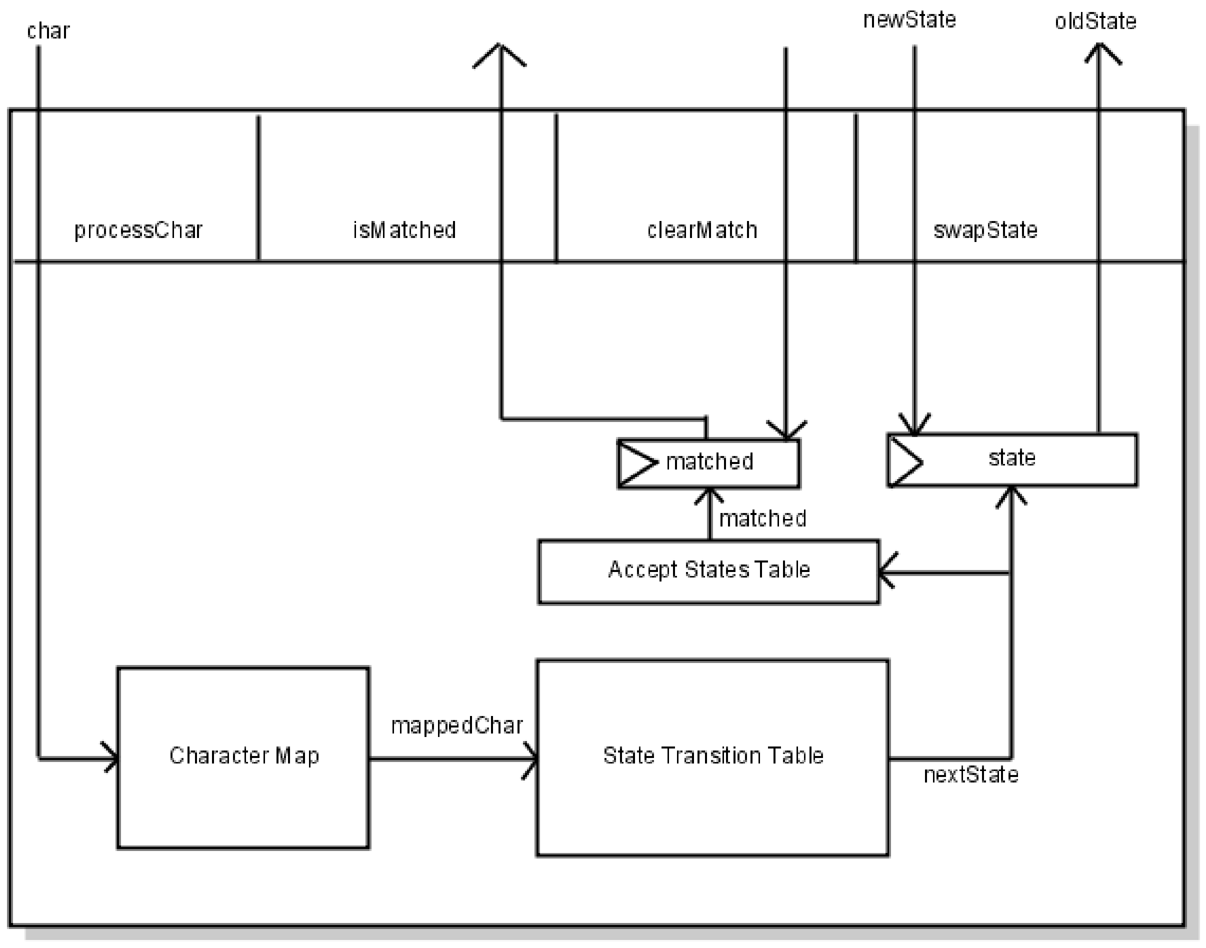
A controlling FSM coordinates the operation of the sub-modules and preserves ordering of the data. Once the headers are hashed, packet data is buffered internally so that the next headers can be hashed while the hash table operation occurs. The controlling FSM generates a ChangeConnection system packet with the correct connection ID for each TCP packet. If the connection is not present in the hash table, the FSM also generates a NewConnection system packet.

Core Logic - Regular Expression Matcher

The regular expression matcher supports parallel matching of up to 64 regular expressions at speeds of up to 800Mbit/s (running at 100Mhz). Regular expression matching is performed on connection streams, not individual packets; therefore, the regular expression matcher must maintain state for each connection between packets from that connection. Providing 8 bits of state (described below) for 65,536 connections requires 64KB of state per regular expression, for a total of 4MB for 64 regular expressions, which is stored in DRAM using the memory subsystem.

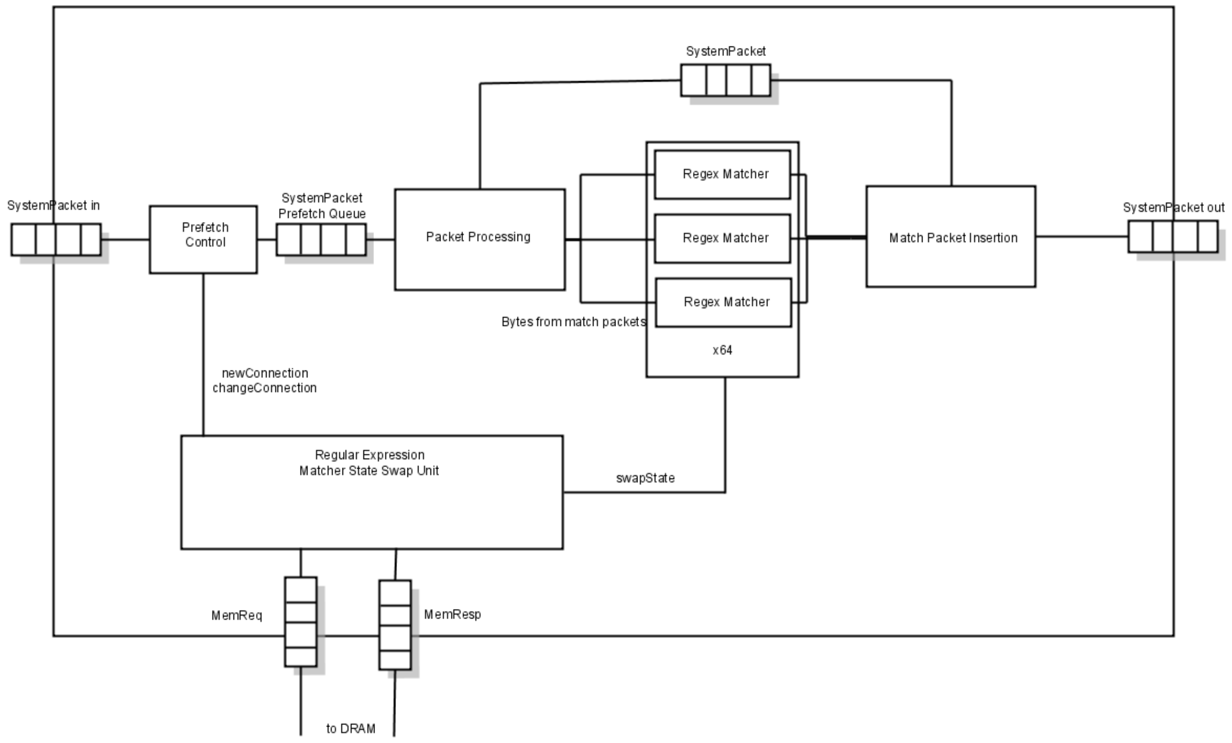
Two approaches are commonly used for regular expression matching in hardware: DFAs, and NFAs with a one-hot encoding [1]. DFAs are essentially finite state machines, so they can be easily implemented in hardware with a state register and a state transition table (i.e. LUT). NFAs are like finite state machines, except that an NFA can be in multiple states at once (this is why the one-hot encoding for state is required). In most cases, the DFA and NFA for a regular expression will have roughly the same number of states; therefore, due to the one-hot encoding, NFAs require an exponentially higher number of bits of memory than DFAs for a given regular expression. Since the cost of storing regular expression state is multiplied by the number of connections (64K), using a DFA was clearly a better choice.

A regular expression is matched by a Regex Matcher, which is illustrated below. A Regex Matcher contains 8 bits of state, which stores up to 128 DFA states (1 bit is used to turn off the matcher if the connection matched in the past). Characters are passed into the Regex Matcher, and pass through a character map (LUT), which maps the 256 ASCII characters down to a smaller set, and then through the state transition table (LUT), which determines the next state. The next state is also passed through a table of accept states (LUT) to determine if a match has occured. The Regex Matcher has two methods, isMatched and clearMatched; when a match occurs, no more characters can be processed until clearMatch is called. This ensures that the system has a chance to detect the match before continuing to process more payload data. Finally, the swapState method inserts new state into the Matcher (i.e. when a packet arrives from a different connection) and returns the old state for storage.



The Character Map, State Transition Table, and Accept States Table LUTs for each of the (up to 64) Regex Matchers are automatically generated during the build process from a list of regular expressions. We use JLex, a lexical parser, to produce DFAs for each of the regular expressions. The output of JLex (a java source file) is then parsed by our script to extract the DFA LUTs and generate bluespec functions for these LUTs.

The Regex Unit is the top-level module in charge of regular expression matching. It is connected to the rest of the pipeline with SystemPacket FIFOs; the Regex Unit inserts Match packets into the SystemPacket stream when matches have occurred. SystemPackets are first processed by the Prefetch Control, which activates prefetch of Regex Matcher state from DRAM (mediated by the State Swap Unit). The SystemPacket Prefetch Queue can be long, which allows the State Swap Unit enough time to fetch the state for the 64 Regex Matchers from DRAM before the new TCP packet reaches the Regex Matchers. When the new packet does arrive, a swapState occurs, which puts the new state into the Matchers and stores the old state back in DRAM. When a match does occur, it is detected by the Match Packet Insertion rule, which inserts a match packet. If detecting a match among all of the 64 Matchers is too slow, we planned to cycle between the Matchers or use a binary tree approach, but we did not run into this problem.



Core Logic - Backlog Buffer Unit

The backlog buffer unit is responsible for (a) filtering the incoming SystemPacket stream so that only matching connections are sent to the UDP sender; and (b) storing (and updating) a 2KB 'backlog' of each connection stream, which is flushed when a match occurs.

The backlog buffer stores the 2KB backlog for each connection in a circular buffer in DRAM. Pointers to the start and end of these buffers are also stored in DRAM. The backlog buffer unit uses 2KB * 65536 = 128MB of DRAM for the backlog (plus a small amount for the pointers). The unit also stores a table of which connections have been matched; these are not stored in DRAM, in order to reduce latency.

When a new TCP packet (from an unmatched stream) enters the backlog buffer unit, an FSM loads the pointers to the start and end of the circular buffer from RAM (and stores the pointers from the previous packet connection). This can be 'prefetched' from the DRAM to avoid throughput reduction. Then, payload data is written to the circular buffer (overwriting old data after 2KB). At the end of the packet, the pointers are written back to RAM. When a match packet enters the unit, an FSM flushes first the connection header information (i.e. IPs, ports, which regex was matched, etc.), and then the KB backlog from DRAM. Once this data has been flushed, the rest of the packet payload is passed through to the UDP sender.

When a TCP packet from a matched stream enters the unit, it is simply passed through to the UDP sender. The match table in the unit (stored in registers) keeps track of which connections have been matched, so the unit knows which packets to pass through and which to store in the backlog buffer.

Core Logic - UDP Sender
The UDP sender is responsible for generating UDP packets that can be sent out over the wire by using the Ethernet MAC. These packets are hard coded to have a fixed source and destination IP and MAC address for simplicity. In a slightly more advanced implementation, these values could be pulled from configuration registers.

The UDP sender sends two types of packets, Match packets and Data packets. Match packets simply indicate that a match has occurred in a given connection, and gives the information for that connection (source and destination IP and ports) as well as the index of the matching regular expression. Data packets give the same connection information, as well as a chunk of data from the stream.

IP packets contain a header checksum which is not optional - it is calculated from the contents of the IP header in the outgoing packet. The method of calculation is by adding all of the 16-bit words in the header together by ones complement addition. Ones complement addition is almost the same as normal addition, but with the caveat that an overflow requires that one be added to the result.

UDP packets contain an additional checksum, but this was not used in our system. Instead it is set to all 0, which indicates that it is not used.

Packets are generated in one of four cases:

1. A match occurs
2. A change connection occurs (new packet of a different connection arrives at input)

3. Maximum packet length is reached
4. A ~40ms timer, which is reset every time payload data enters the UDP sender, expires

One limitation of using the UDP protocol is that it does not have any built in mechanism for ensuring delivery of all packets. While in our controlled environment packet losses were minimal (and aided by Ethernet Flow Control), they were still nonzero. Probably the answer is to use a more sophisticated protocol, or perhaps a different mechanism for forwarding matched streams altogether. Making the system interface via PCIe, for example, might be an appropriate alternative.

Microprocessor Subsystem
A Xilinx Microblaze processor is present in our hardware system and is used for two purposes. The first is initializing some registers in the Xilinx Ethernet MAC core on startup, and the second is to retrieve debugging and statistical information from the core logic and send it over an RS232 port for analysis. A CoreConnect PLB bus system running at 125MHz was used to connect the Microblaze processor to its own instruction and data memory (both stored in BRAM) as well as the RS232 UART, the DDR2 SDRAM, the Ethernet MAC, and a special PLB port exported from the core logic.

Inside the core logic, a CBus was instantiated and connected to this PLB interface. Several statistics counters were scatters across the core logic and placed on this CBus.

Ethernet MAC Subsystem
The Xilinx XC5VLX110T has two hard ethernet MAC cores embedded in silicon. We used one of these for our project. To interface with the hard core, we used the Xilinx XPS_LL_TEMAC wrapper IP. This wrapper exposes two important interfaces - a PLB slave interface, which is used by the microprocessor to initialize various configuration registers, and a Xilinx LocalLink interface, which is used to interface with our core logic.

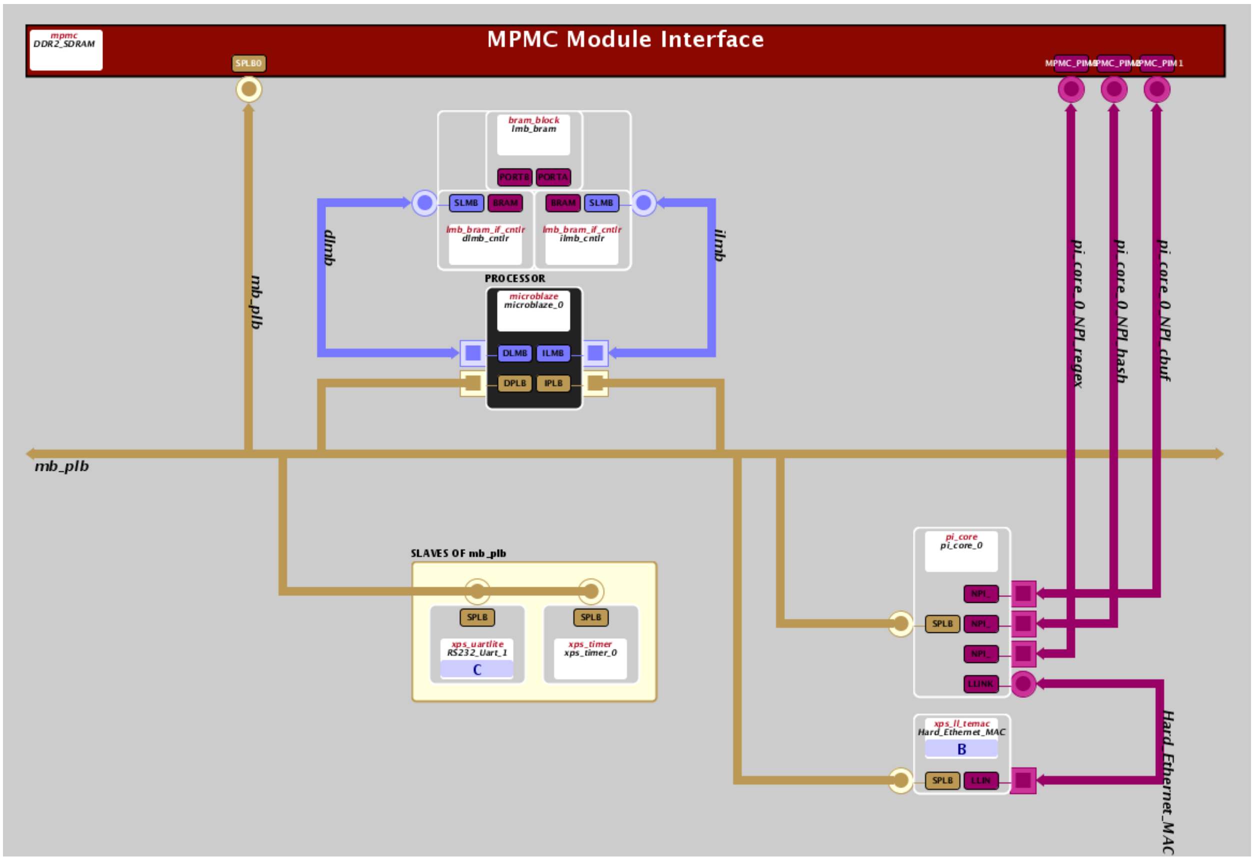
The LocalLink interface consists of one transmit and one receive port, which very much resemble Bluespec Get/Put interfaces. An initiating LocalLink component provides a "SrcRdy" signal, and a target LocalLink component provides a "DstRdy" signal. When both of these signals are asserted, data transfer takes place. LocalLink transfers "packets" of data, which contain a fixed number of header and footer words, with a "payload" between them. In our case, the payload is simply a decoded Ethernet frame, and the header and footer words are unused (though they still must be transferred across the LocalLink).

As mentioned earlier, this LocalLink interface is easily wrapped up into Bluespec Gets and Puts for connection to the core logic.

One frustration we ran across in hardware is that the XPS_LL_TEMAC IP actually doesn't meet the LocalLink specification in the TX path. In order to function correctly, it is mandatory that a two-cycle stall be inserted between the sending of the LocalLink header, and the LocalLink payload, even though the DstRdy signal is asserted. Xilinx has documented this problem in Answer Record #32317. Unfortunately, no link to said errata is provided on the product page where the (incorrect) documentation is provided. The only way to find it was by searching for XPS_LL_TEMAC and scouring through about 5 pages of mostly unrelated results. Others have described the Xilinx website to us as "the world's greatest scavenger hunt," we guess they weren't kidding.

Using the Xilinx EDK

As our project uses several Xilinx EDK components (Microblaze, MPMC, Ethernet MAC), we had to interface with the Xilinx EDK infrastructure. The path we chose was to build our core logic into a Xilinx EDK "peripheral" which could then be connected to the rest of the EDK system through the Xilinx Platform Studio GUI program.



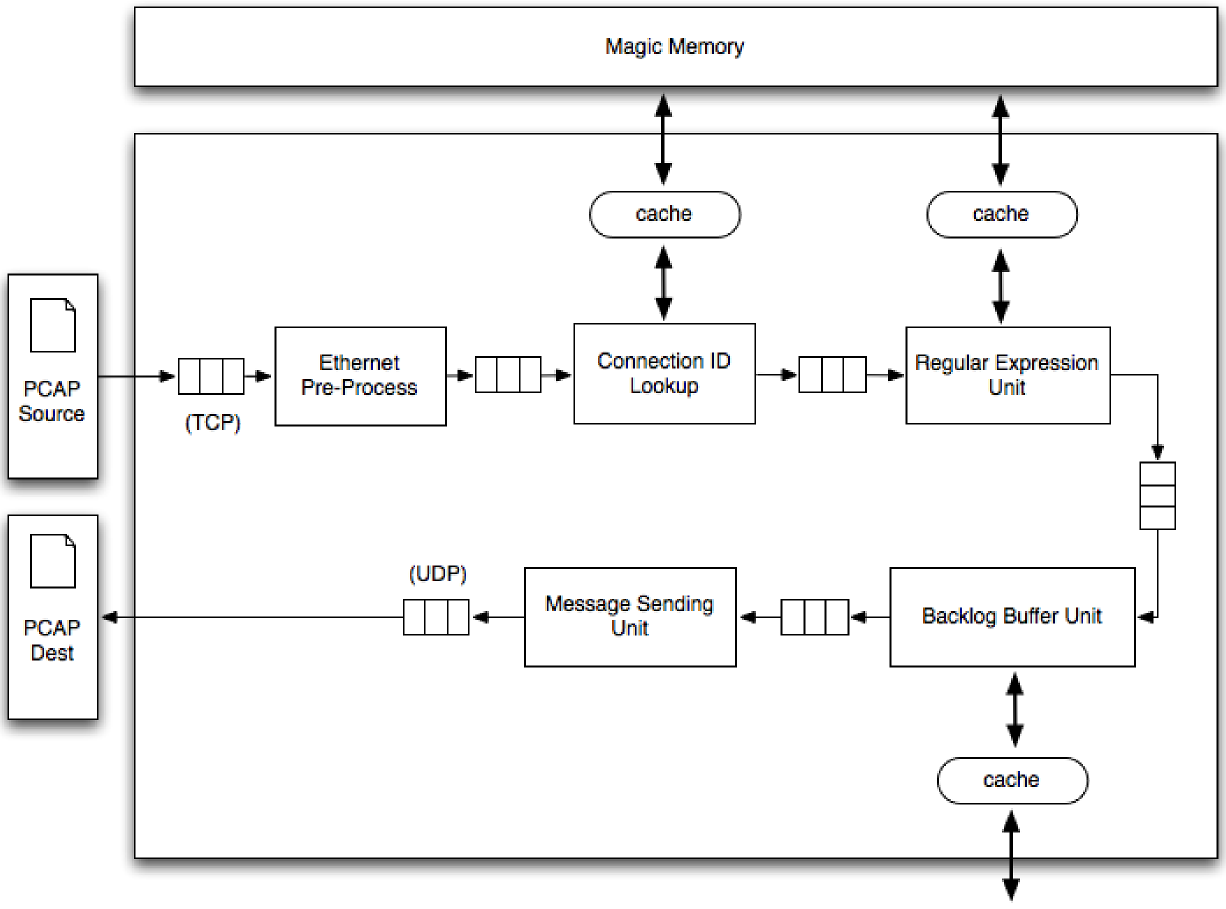
In order to do this, we used Synplify Pro to create an EDIF-format netlist of our synthesized design. We then created a top-level Verilog wrapper which contained a black box instance of our core logic netlist. This top-level wrapper also inverted reset polarity, as the EDK system uses active-high resets, where Bluespec synthesized Verilog uses active-low resets. The top-level wrapper also included an instantiation of the Xilinx PLBV46_SLAVE_SINGLE IP, which is a Xilinx VHDL core that provides a bridge between the PLB bus used in the Xilinx EDK system, an a simpler Xilinx IPIC interface, which is exported from the core logic to provide access to the internal CBus.

With all this in place, three special files that describe the EDK component were created. A black box description (BBD) file, a microprocessor peripheral description (MPD) file, and a peripheral analyze order file (PAO). These files, along with the core logic netlist and the wrappers mentioned above were placed in a Xilinx EDK component repository, so that they could be accessed by Xilinx Platform Studio.

Of the three files, only the MPD file is particularly interesting - the other two only list paths of the netlist (BBD) and HDL (PAO) files used by the peripheral. The MPD file defines all of the ports in and out of the peripheral. In our case, this was a PLB slave port, three NPI ports, and a LocalLink port, as well as three sets of clock/reset lines (one for the core logic, one for the PLB/LocalLink bus, and one for the memory interface). The PLB/LocalLink and the NPI clocks were run at 125MHz, the default for an EDK system with our FPGA. The core clock was run at 100MHz in order to make timing.

Testing Framework

To run end-to-end system tests we created a module that read and wrote packets to a PCAP file, a standard format for representing packet data used by packet sniffers. This allowed us to capture packets from the wire, run them through our simulation, and view the output in the *Wireshark* packet sniffer. We replaced the DDR2 with "magic" memory in our test harness.



Results
We synthesized our project on a Xilinx XC5VLX110T FPGA. Though we were close to meeting our 125MHz timing goal, we reverted to a much more conservative 100MHz clock for our core logic, which we met easily. To interface with the Xilinx IP running at 125MHz, several synchronizing FIFOs were used to communicate with the NPI, LocalLink and PLB interfaces.

Once correct behavior was verified in hardware, we wanted to get an idea of our system's performance. At first, we simply tried generating packets on a PC with a Gigabit Ethernet NIC and sending them as fast as possible. This proved to be inconclusive, however, because none of the PCs we used were able to saturate the Gigabit link, and the throughput out of the PC was highly dependent on the packet size of the test vector. Because we expected a relationship between packet size and throughput in our system as well, we needed to control for this in the test source.

Instead, we adopted a hardware-based solution. A special StreamCaptureFIFO was used in order to push data through the system as fast as possible. The StreamCaptureFIFO buffers up 256KB of incoming data from the Ethernet link (and does not allow any data to be dequeued). Once the StreamCaptureFIFO is full, it allows data to be dequeued (but not enqueued). Once the StreamCaptureFIFO is empty, the process starts all over again. This allowed for 256KB chunks of the stream to be pushed through the system as fast as possible. We then counted the bytes as they were pushed through, along with the number of cycles that the StreamCaptureFIFO was in "Draining" mode, and used this to create our benchmark.

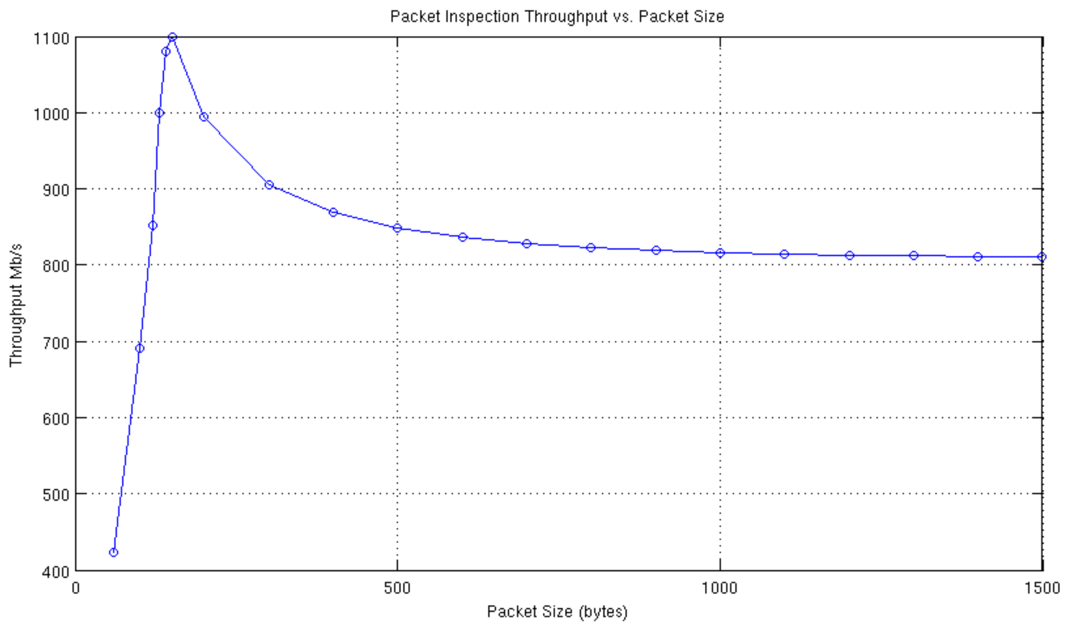
We used three test vectors - each a sequence of several thousand packets of a fixed size, each having a unique source and destination IP and port number. This means that each packet would cause the allocation of a new connection. The results were as follows:

Packet Size	**System Throughput**
1500 bytes	522 Mbps
500 bytes	522 Mbps
100 bytes	547 Mbps

Knowing that the Regular Expression Unit running at 100MHz should be capable of around 800Mbps throughput, we were a bit surprised by these results, especially in light of the fact that the throughput *increased* with a smaller packet size. We had been expecting the opposite, as smaller packets require more frequent memory accesses, which, we believed, would be the limiting factor. That the performance increased suggested that something not related to memory was the culprit.

After some investigation, we determined that the source of the problem was a rule conflict within the Regular Expression Unit. The result was that processing 4 bytes of data was taking 6 clock cycles, instead of 4 clock cycles. This comes out to a maximum throughput of around 508Mbps, which is much more in line with what we were observing. This rule conflict was fixed, and the tests run again, yielding much better results.

Packet Size	**System Throughput**
1500 bytes	811 Mbps
1400 bytes	811 Mbps
1300 bytes	812 Mbps
1200 bytes	813 Mbps
1100 bytes	814 Mbps
1000 bytes	816 Mbps
900 bytes	819 Mbps
800 bytes	823 Mbps
700 bytes	828 Mbps
600 bytes	836 Mbps
500 bytes	849 Mbps
400 bytes	869 Mbps
300 bytes	906 Mbps
200 bytes	994 Mbps
150 bytes	1100 Mbps
140 bytes	1080 Mbps
130 bytes	999 Mbps
120 bytes	852 Mbps
100 bytes	691 Mbps
60 bytes	423 Mbps



The plot of throughput is corresponds very well to the expected behavior. Small packets show a marked and sudden breakdown in performance below a certain threshold. This is the point where a memory access can no longer complete in the time it takes to push a single packet through the system, and limits the rate of packets through the system.

The more gradual decay as packet size increases can be explained by the fact that each packet has a certain amount of overhead contained in the Ethernet, IP, and TCP headers (54 bytes in our test). The Regular Expression Unit only processes the payload of these packets. The headers are discarded in the Ethernet Preprocessor. For streams of smaller packets, the percentage of overhead that is discarded prior to the Regular Expression Unit is much higher, meaning that the Regular Expression Unit has to do less processing to maintain the same level of system throughput. For example, a 200 byte packet is 25% overhead. Only 150 bytes of it actually need to be processed by the Regular Expression Unit.

Citations

[1] Moscola, J.; Lockwood, J.; Loui, R.P.; Pachos, M., "Implementation of a content-scanning module for an Internet firewall," *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on* , vol., no., pp. 31-38, 9-11 April 2003
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1227239&isnumber=27544