NAT implementation for the NetFPGA platform

Omar Choudary
University of Cambridge
15 JJ Thomson Avenue
Cambridge, UK
osc22@cl.cam.ac.uk

David J. Miller
University of Cambridge
15 JJ Thomson Avenue
Cambridge, UK
david.miller@cl.cam.ac.uk

ABSTRACT

We present an implementation of NAT (Network Address Translation) for the NetFPGA platform capable of line-rate Gigabit Ethernet. Our implementation features RAM and CAM (Random Access and Content Addressable) memories for a fast and efficient NAT table. Several simulation and regression tests are included.

1. INTRODUCTION

The NetFPGA platform [4] is an open platform for research and education in building Gigabit Internet routers and similar devices. As part of the "Building an Internet Router" (P33) module (part of the M.Phil ACS course at the University of Cambridge), we were required to build a working IPv4 router, and then develop an "advanced feature". The students of this class were grouped in teams of two or three students each, in which one was in charge of hardware design while the other(s) developed software.

Even though a fully implemented reference router is available for the NetFPGA, we had no access to its implementation, and were required to design and implement the core functionality on our own. Only the basic reference pipeline, which provides access to the physical Ethernet ports and CPU registers, was provided. In addition, we were required to implement our own longest prefix match look-up table, rather than use the Xilinx CoreGen (T)CAM modules (which were forbidden).

After building the basic router, we began the work on our advanced feature. We chose to implement Network Address/Port Translation (NAPT, also known as traditional NAT) [6], henceforth called NAT. The desired functionality was that of a home router, but allowing the greater flexibility of supporting multiple Internet interfaces (*i.e.*, a multi-homed router). In this paper, we document the design of our NAT implementation and the simulation and regression tests provided.

Like the reference router, our solution achieves line speed, *i.e.*, close to 1 Gbps. This performance was achieved by implementing our own Content Addressable Memory (CAM), based on FPGA BlockRAMs. A CAM is used to locate the information required to translate private addresses and ports of each packet crossing the router. The BlockRAMs available in the Xilinx Virtex-II FPGAs offer a large block of dual-ported memory, which makes them convenient for use as a CAM. Performance measurement was done using TTCP [5] between two PCs connected via the NetFPGA board.

2. IMPLEMENTATION

This section describes our implementation, which is available on-line [1].

2.1 Overview

The NetFPGA platform provides a basic framework, known as the reference pipeline, to build Internet devices such as switches and routers. Packets arriving at an Ethernet port pass through the pipeline, where they can be modified across several modules, before being transmitted via the same or a different port. In this architecture, the NetFPGA board is used to process most of the packets, while software running on the host PC (connected to the NetFPGA board via PCI bus) is used to handle exceptions (including corrupt packets, ARP requests, etc.), process new flows, and update the translation table within the NetFPGA via PCI registers.

The overview of our system design can be seen in Figure 1. The IP Filter module is in charge of all layer 3 (routing) and layer 4 (translation) operations. The arrows show the flow of data across the packet data path and register bus. The packet data bus is used to transfer the packets between different modules using a 64-bit data path. Our implementation uses a pipeline to provide sub-modules (ARP, IP Filter, routing, and NAT) with access to required header fields without sacrificing throughput, but at the expense of a few clock cycles of latency. Pipelining makes it possible to begin receiving a new packet while the last packet is still being processed.

Our hardware NAT implementation is done in the NAT Table module. This module is used to perform layer 3 and 4 address translation for hosts behind the NAT-enabled interfaces, as described in the next section.

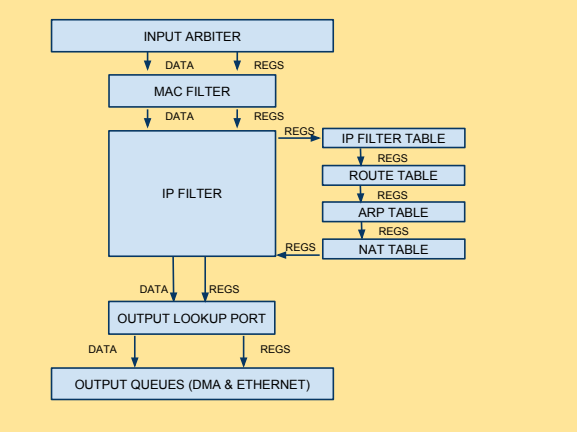

Figure 1: ***NAT design overview***

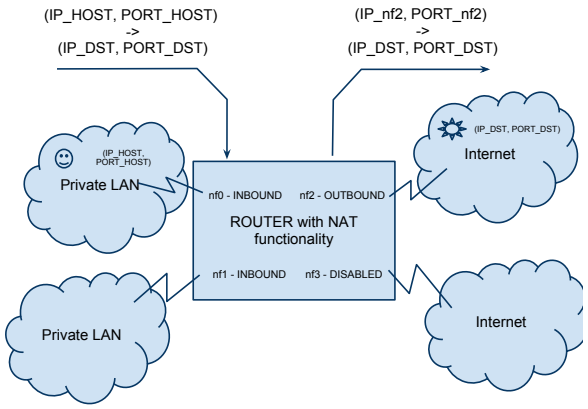

Figure 2: ***NAT process using the NetFPGA framework***

2.2 NAT design

The NetFPGA hardware has four Ethernet ports. For our NAT implementation, we allow each port to be in one of three states: `INBOUND`, `OUTBOUND` or `DISABLED`.

The `INBOUND` state is used strictly for NAT. That is, a packet arriving on a `INBOUND` interface will only be checked against the NAT table. When a match is found, the packet will be forwarded according to the result of the NAT match. When no match is made, the packet will be sent to the host software. This consideration is taken because we need a way to know when a new entry is needed in the NAT table. Thus, new entries in the NAT table are created ONLY when a packet arrives on a `INBOUND` interface, for which a match is not found in the NAT table. If we were to enable routing on an `INBOUND` interface, we would have no way of knowing when a new NAT entry is needed.

Interfaces in the `OUTBOUND` state are used for both NAT and routing. When a packet arrives on an `OUTBOUND` interface, it is first checked (in the IP Filter Table module) to see whether the destination address matches one of the addresses of the router. If the packet is addressed to our router, then we attempt to match against the NAT table. If the packet is not for the router (*i.e.*, is to be routed), then it is checked against the route table as usual.

Interfaces in the `DISABLED` state are used only for routing. That is, a packet arriving on such interfaces will never be checked against the NAT table. An example of a possible configuration is shown in Figure 2. In this example, a packet is sent from a host on the private LAN (RFC1918 network) to a public destination. The NAT implementation replaces the source IP address and port with the IP address of the `OUTBOUND` interface, and the source port with a locally allocated port.

2.3 NAT table

The core of our solution is the NAT table. This table is implemented using a combination of RAMs and CAMs. The RAM is implemented using simple SelectRAM, while our CAMs are implemented in BlockRAMs. Our CAM implementation is based on Xilinx Application Note 260 [7].

The CAM is used to locate existing flows in the NAT table. If an entry is found, the RAM provides the information required to translate the private network address, before the packet is forwarded. To check against both `INBOUND` and `OUTBOUND` interfaces, we use two different tables. The information stored in both CAM and RAM is shown in Table 1.

The RAM and CAM are bundled together into a module named `RAM_CAM_128_96`. The RAM (`INIT_RAM32x128`) has 32 entries of 128 bits each, implemented using four 32x32 SelectRAMs. The CAM (`CAM_96`) has 32 entries of 96 bits each, implemented in 11 BlockRAMs. Each BlockRAM in the Xilinx Virtex-II family — used by the NetFPGA — has a capacity of 16 kilobits, providing for a CAM with a 9 bit search input and 32 entries.

`RAM_CAM_128_96` completes a search in one clock cycle, and returns the search result (signal `match_ok`), along with the associated data (if any) in one more cycle. The diagram of this module is shown in Figure 3.

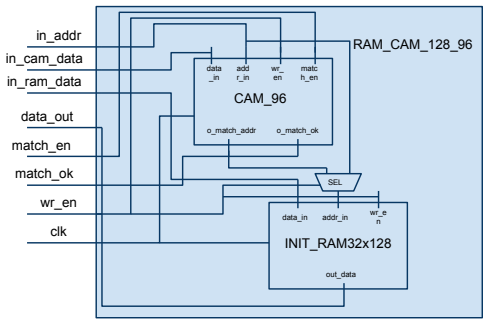

Figure 3: ***Diagram of the RAM_CAM_128_96 module***

Table 1: Contents of RAM and CAM for NAT Table

Incoming Interface	CAM data (96 bits)	RAM data (128 bits)
INBOUND	IP_HOST, IP_DST PORT_HOST, PORT_DST	IP_NAT_OUT, PORT_NAT_OUT IF_NAT_OUT, MAC_NEXT_HOP
OUTBOUND	IP_DST, IP_NAT_OUT PORT_DST, PORT_NAT_OUT	IP_HOST, PORT_HOST IF_NAT_IN, MAC_HOST

Module NAT Table is implemented using two `RAM_CAM_128_96` modules. The first instance is used to match packets arriving on an `INBOUND` interface, when the second instance is used for packets arriving on an `OUTBOUND` interface.

2.4 Layer 4 processing

Strictly speaking, a router requires only layer 3 address information in order to make routing decisions. It has no need to decode any layer 4 protocol present, and therefore need not check any layer 4 checksum such as found in TCP and UDP.

In the case of a translating router, layer 4 port information is both used to make decisions, and rewritten before the packet is forwarded. It should therefore check layer 4 checksums before translation in order to prevent damaged packets from being mis-translated, mis-routed, or filling up the translation table with non-existent flows.

Given the time constraints of the course, we elected not to check layer 4 checksums. In the event of a damaged packet, any recipient of the packet will drop it, and the applicable layer 4 error detection mechanism will cause the packet to be retransmitted. Our NAT router is prone to overflow of its translation table by damaged packets (a potential form of denial-of-service attack), but this vulnerability could be rectified by implementing a checksum check before the translation module. Note that because the PCI bus has less bandwidth than the network links, this check must be done in hardware, lest a flood of damaged packets overload the PCI bus.

Since translation involves rewriting layer 3 and layer 4 headers, it is necessary to update the TCP/UDP checksum before forwarding. We can do this without recomputing the checksum over the entire packet by applying the formula:

$$H' = \sim(\sim H \oplus \sim sum_{old} \oplus sum_{new})$$

where H is the old checksum, H' is new checksum, sum_{old} and sum_{new} are the ones'-complement sum computed over `SRC_IP`, `DST_IP`, `SRC_PORT` and `DST_PORT` of the old and new values respectively (the only fields that change their value). $\sim$ indicates a bit-wise complement operation, and $\oplus$ indicates a ones'-complement sum operation.

In order to free memory in the NAT table for new connections, the host software deletes NAT table entries at regular intervals. However, when two hosts A and B communicate using TCP, they use the same TCP port for every packet they send. If the software deletes the entry for such TCP communication, then the next packet of A towards B will have a different port and B will reject it. For this reason entries in the NAT table that refer to active TCP connections should not be removed unless they are no longer active. We provide a hit-counter for each NAT table entry, implemented in a SelectRAM of 32 entries of 32-bits each, which is incremented every time an entry is matched. The host software can use this to detect and delete inactive flows.

2.5 Module interface

The NAT table is implemented in the module `nat_top`, which instantiates the two `RAM_CAM_128_96` modules. We designed `nat_top` such that it can be used with any NetFPGA design. It accepts, as input, the IP source, IP destination, source port, destination port, input and output interfaces. It provides, as output, a signal that indicates whether a match was made, and the required data for translation (IP, port, interface and MAC).

2.6 Software communication

While the hardware performs translation and forwarding of known flows, it depends on host software to set up new flows in the translation table, and to remove flows once they expire. Configuration of the translation table is performed by means of NetFPGA PCI registers. The registers used by our NAT implementation, and their description, are presented in Table 2.

3. EVALUATION

In order to assess the correct functionality of our NAT implementation, we followed the regression test driven approach suggested by Covington *et al.* [3]. We created several such simulation and regression tests. Since our NAT module is part of the standard router data path (see Figure 1), it was also tested against the simulation and regression test suites provided with the reference router. All these tests passed successfully.

3.1 Verification by simulation

For simulation, we designed the tests `test_nat_tcp` and `test_nat_udp`, each of which send three packets between an `INBOUND` and an `OUTBOUND` interface, in order to test address translation in both TCP and UDP. The first packet is sent from `OUTBOUND` to `INBOUND` to check that the packet is routed instead of translated, since there is not yet a matching entry in the NAT table. Second packet is sent from `INBOUND` to `OUTBOUND` and should be translated appropriately. The third packet is again sent from `OUTBOUND` to `INBOUND` to verify that it is properly translated. The tests populate all the tables with the necessary data.

3.2 Verification by regression

We developed two regression tests named `test_nat_setup` and `test_nat_udp`. The former automatically populates the

Table 2: Registers used by the NAT implementation

Register	Description
ROUTER_OP_LUT_NAT_IFACE_MODE	0 = DISABLE, 1 = INBOUND, 2 = OUTBOUND
ROUTER_OP_LUT_NAT_IFACE_WR_ADDR	change the NAT mode of selected interface
ROUTER_OP_LUT_NAT_IFACE_RD_ADDR	get the NAT mode of the specified interface
ROUTER_OP_LUT_NAT_TABLE_ENTRY_IP_HOST	IP address of HOST at INBOUND interface
ROUTER_OP_LUT_NAT_TABLE_ENTRY_IP_DST	IP address of DST communicating with HOST
ROUTER_OP_LUT_NAT_TABLE_ENTRY_PORT_HOST	TCP/UDP source port used by host
ROUTER_OP_LUT_NAT_TABLE_ENTRY_PORT_DST	TCP/UDP destination port used by destination
ROUTER_OP_LUT_NAT_TABLE_ENTRY_IP_NAT_OUT	IP on OUTBOUND interface towards DST
ROUTER_OP_LUT_NAT_TABLE_ENTRY_PORT_NAT_OUT	TCP/UDP port allocated by the software
ROUTER_OP_LUT_NAT_TABLE_ENTRY_IF_NAT_IN	NAT INBOUND interface
ROUTER_OP_LUT_NAT_TABLE_ENTRY_IF_NAT_OUT	NAT OUTBOUND interface
ROUTER_OP_LUT_NAT_TABLE_ENTRY_MAC_HOST	MAC address of host
ROUTER_OP_LUT_NAT_TABLE_ENTRY_MAC_NEXT_HOP	MAC address of next hop towards destination
ROUTER_OP_LUT_NAT_TABLE_ENTRY_COUNTER	used to maintain active connections
ROUTER_OP_LUT_NAT_TABLE_RD_ADDR_REG	entry index for read
ROUTER_OP_LUT_NAT_TABLE_WR_ADDR_REG	entry index for write

NAT table (which ordinarily would be done by the host software) and tests the NAT implementation by transmitting packets between two PCs connected via the NetFPGA's Ethernet ports. The latter is a hardware replica of the simulation test of the same name. A similar test for TCP packets could easily be developed, however we successfully tested the NAT implementation with TCP by using web-based applications such as YouTube and Google Maps.

For both simulation and regression tests, we use the Perl libraries provided in the NetFPGA base kit, along with a library of our own.

3.3 Performance

We measured the maximum throughput of our solution using the Test TCP benchmarking tool between two hosts connected via the NetFPGA board. Host A was connected to an `INBOUND` interface, and host B to an `OUTBOUND` interface.

While transferring data from A to B, our router checked each packet against four tables: the IP Filter (to test whether the packet is for this router), the NAT table (to find what translations need to be done), the routing table (to find the next hop), and finally the ARP table (to find the MAC address of the next hop). Our solution managed to achieve this chain of look-ups at line speed, with TTCP reporting approximately 133 MB/s, indicating a fully saturated 1 Gbps link.

4. FURTHER WORK

While we provide a suite of simulation and regression tests, these could be expanded to use the FPGA packet generator [2] to verify the maximum capabilities of our solution.

Additional robustness could be added by providing layer 4 checksum checking. Additional features may be added, such as ICMP translation, or the ability to mark certain flow types as always requiring host processing, to enable the support for protocols that require special handling (such as FTP).

5. CONCLUSION

In this paper, we presented a line-rate (1 Gbps) NAT implementation for the NetFPGA platform. We describe our implementation and its design, including how FPGA memory resources are used to construct a fast look-up mechanism.

As is convention, we developed several simulation and regression tests for our solution to verify its correctness. Additionally, our NAT module passes the regression tests provided with the reference router.

We hope this solution may be of use to others in the NetFPGA community.

6. REFERENCES

[1] O. Choudary. Source code of the NAT implementation for the NetFPGA platform. `http://www.cl.cam.ac.uk/~osc22/p33nat/`.

[2] G. A. Covington, G. Gibb, J. W. Lockwood, and N. McKeown. A packet generator on the NetFPGA platform. *17th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2009.

[3] G. A. Covington, G. Gibb, J. Naous, J. W. Lockwood, and N. McKeown. Encouraging Reusable Network Hardware Design. In *NetFPGA community*, 2009.

[4] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. Mckeown. NetFPGA: An open platform for teaching how to build gigabit-rate network switches and routers. *IEEE Transactions on Education*, 2008.

[5] PCAUSA. Test TCP (TTCP) utility. `http://www.pcausa.com/Utilities/pcattcp.htm`.

[6] P. Srisuresh, J. Networks, and K. Egevang. RFC3022: Traditional IP network address translator (traditional NAT). `http://tools.ietf.org/html/rfc3022`.

[7] Xilinx. XAPP260: Using Virtex-II BlockRAM for high performance read/write CAMs. `http://www.xilinx.com/support/documentation/application_notes/xapp260.pdf`.