Project Report – Group 7

Network-attached Encryption Accelerators

by

Chun Yee Chu
Serdar Ozturk
Soheil Shahrouz

April 2023

A report submitted in conformity with the course requirement of ECE532

Department of Electrical and Computer Engineering
University of Toronto

Table of Contents

1. Overview

In this section, we first discuss why we chose to work on this project and outline our initial goals. We then present a high-level block diagram of the system and provide an overview of the system's general functionality and each block's specific function.

1.1 Motivation

As the project title suggests, our goal was to design and implement power-efficient encryption accelerators capable of low-latency encryption of network data. While the same functionality can be achieved using a pure software-based approach, a software-only implementation has some characteristics that are not desirable:

In a software approach, the operating system manages the network interface. Therefore, if a process wants to receive or transmit data through the network interface, it cannot interact with the network interface directly. Instead, it must copy data to or from the kernel space, which manages the network interface. These redundant copies require the processor to move data into an external memory. However, using an external memory has three disadvantages: First, it introduces high latency to the application. Second, the added latency is not deterministic. Third, since most of the energy in a computer system is consumed by the memory hierarchy, such a software-based approach is power inefficient.

To address the aforementioned concerns, we have proposed the utilization of network-attached encryption accelerators implemented on an FPGA board. Our approach involves the complete implementation of the network stack through the use of customized IP-cores in hardware. The design employs a dataflow architecture that obviates the necessity of external memory by enabling network data to be transmitted between cores in a streaming manner. The elimination of external memory ensures deterministic low latency, and enhances the power efficiency of the design.

Our initial objective for this project was to develop two encryption algorithm accelerators on FPGA that could encrypt data at the network line rate. To enhance power efficiency even more, we planned to use partial reconfiguration for switching between these two encryption algorithms. However, we opted to substitute partial reconfiguration functionality with SD card capability.

1.2 Block Diagram

An overview block diagram of our final design is presented in Figure 1. A legend in the bottom left corner of the figure indicates the cores that we implemented using HLS and the interface types employed to communicate with various cores.

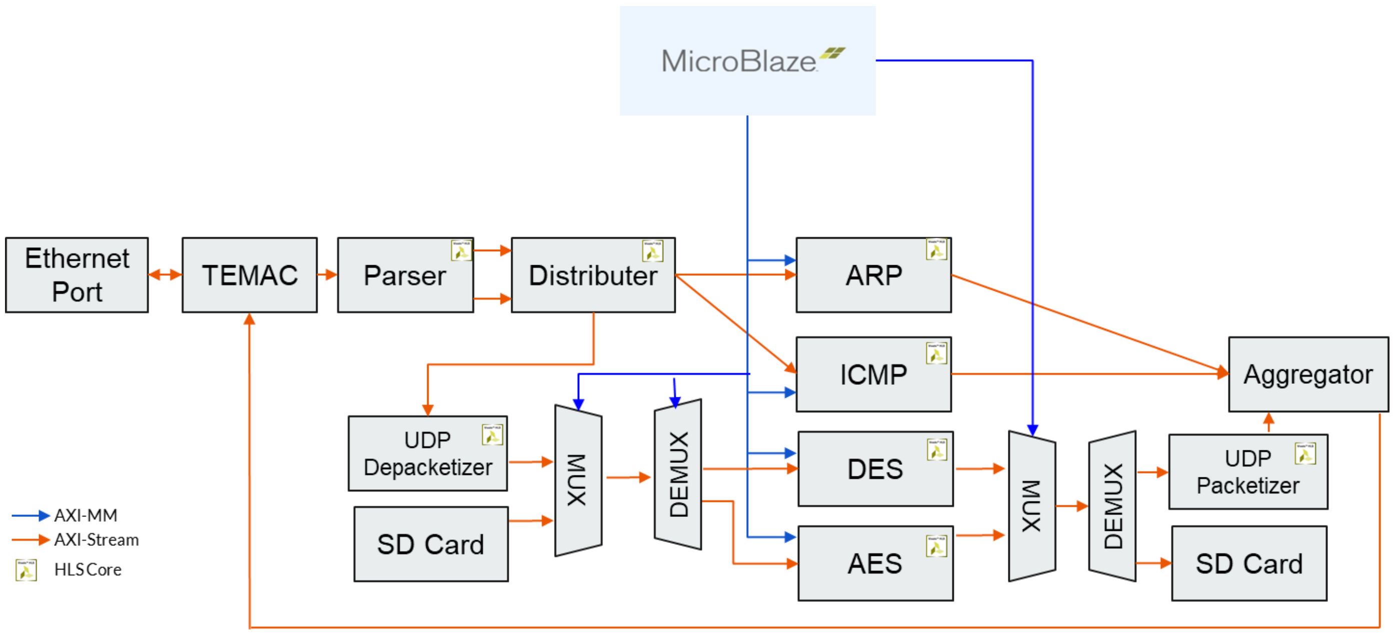

Fig. 1. Block diagram of the final system

The Tri-Mode MAC IP-core from Xilinx was used to establish communication with the PHY chip on the board and handle both the MAC and physical layers. Ethernet frames received by the system are first sent to the parser, which detects the packet type before forwarding it to the distributor. The distributor then routes the packet to the appropriate protocol processor based on the type of packet detected by the parser. For instance, ARP and ICMP processors are responsible for responding to ARP and ICMP requests respectively. The UDP depacketizer, on the other hand, extracts the payload of received UDP packets and passes it to the selected encryption core (AES or DES), which encrypts the data using a specified key. This key can be configured through AXI-Lite interfaces that are controlled by a Microblaze processor. The encrypted data is then received by the UDP packetizer, which puts it into UDP/IP packets and calculates the necessary checksums. Our design features several cores that generate network traffic, and their output AXI-Stream interfaces are all connected to an aggregator, which is actually an AXI-Stream switch. The aggregator interleaves packets from various IP-cores to produce a single AXI-Stream output that is transmitted over the Ethernet port using TEMAC. There is also a SD card IP-core in our design that communicates with the SD card. The system can read raw data from the SD card and pass it to the encryption cores and store the encrypted data back into the SD card.

As specified in Figure 1, network stack and encryption IP-cores are implemented using Vivado HLS. Encryption cores are based on open source software implementations available on Github. Network stack cores are implemented using the protocol descriptions on their corresponding Wikipedia pages. The SD card IP-core is an open source core provided by Digilent. All other IP-cores used in this project were available in Vivado's IP catalog. More detailed information on each block can be found in Section 4.

2. Outcome

2.1 Original System Proposed

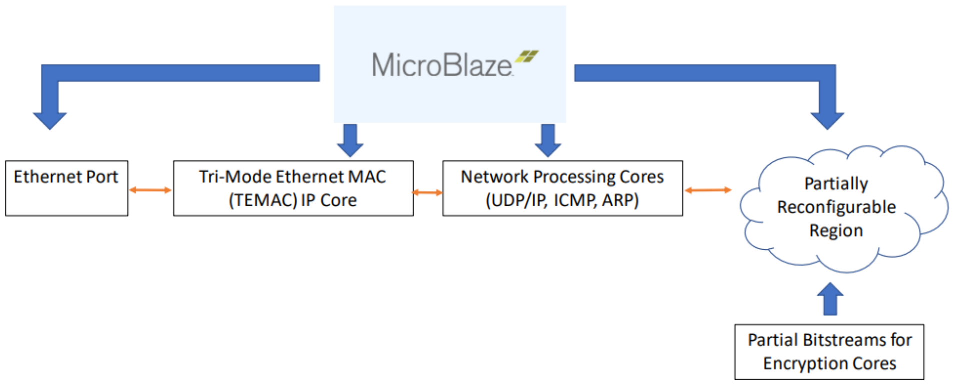
MicroBlaze
Ethernet Port
Tri-Mode Ethernet MAC (TEMAC) IP Core
Network Processing Cores (UDP/IP, ICMP, ARP)
Partially Reconfigurable Region
Partial Bitstreams for Encryption Cores

Fig. 2. Originally proposed block diagram

2.2 Differences Between the Original and the Final System

There are some important differences between the original proposed system and the final system stemming from both limitations and logistics.

Partial Reconfiguration Removed

For the proposed system, encryption cores are partially reconfigurable. For the final project, this function is removed. Logistically, one of our team members left the course halfway through the semester. Thus we decided to reduce the complexity of our project which was already high to compensate for this loss. Practically, partial reconfiguration turned out to be very challenging and full of possible bugs and errors. While solving these bugs is feasible, due to the nature of partial reconfiguration, it meant that we couldn't implement it into our project without fully implementing both encryption cores first. Since each partial reconfiguration project can come with its own set of bugs, this meant we wouldn't be able to start debugging until the tail end of the course. Combined with the correctness concerns that arise from using AXI communication with a reconfigurable region and the fact that it wasn't key to achieve our project goals, we decided to remove it from the project.

SD Card Added

While it wasn't in the proposed system, our final project has an SD card integration where the input to the encryption cores can come from the SD card and their output can be written to an SD card. After removing the partial reconfiguration, we needed to add something in its place to recover some project complexity and populate the remaining milestones. SD card implementation was chosen for this as it increased the already high flexibility of the project. It also solved a possible security concern for the project, as the decrypted data was being transmitted over ethernet, granted over a very short distance. With an SD card, it can be transferred using the card which would improve security.

2.3 Project Evaluation

Most of our planned goals at the beginning of the project were achieved, including smooth handling of data packets by the network stack, and up-to line rate encryptions using the 2 encryption cores. Although partial reconfiguration was not implemented at the end, the encryption process can be successfully done using data from the SD card instead of the network with the correct results written back onto the SD card. Given our change in manpower throughout the project, we would consider the result a success.

2.4 Possible Improvements

- With some better coding and familiarity, SD card read and write speed can be increased.

- SD card integration can be done with Verilog instead of Microblaze to increase speed and reduce the reliance on C code running on Microblaze.

2.5 Next Steps

- Partial reconfiguration can be re-introduced to the project. The encryption cores are used exclusively one at a time. Thus they can be implemented in a partially reconfigurable region, reconfiguring the FPGA during run-time whenever the core to be used switches. This way the resources used on the FPGA would be reduced.
- Other than just using a symmetrical encryption, this project can be developed to also do asymmetrical encryptions. The main difference is that it will involve a public and private key pair for each encryption done, and the benefit is that there can be more use cases for our accelerator.

3. Project Schedule

3.1 Original Milestones Proposed

<u>Milestone 1</u>

- Network stack
    - Read papers on network stack implementation using HLS
- Encryption cores
    - Choose at least 2 encryption algorithms and find their software implementations in C to be used as a baseline
    - Get familiar with HLS by reading tutorials provided by Xilinx
- Partial reconfiguration
    - Learn about the partial reconfiguration concept and figure out if it imposes any limitations on other parts of the project

<u>Milestone 2</u>

- Network stack
    - Use Xilinx TEMAC core to implement the Ethernet protocol
    - Create an Ethernet loop on FPGA

- Encryption cores
    - Implementing part of the algorithms
    - Start considering on testbenches
- Partial reconfiguration
    - Follow a tutorial for Partial reconfiguration through Xilinx

<u>Milestone 3</u>

- Network stack
    - Implement and test ARP protocol on FPGA
    - Implement and test ICMP protocol on FPGA
- Encryption cores
    - Keep implementing and debugging the 2 encryption algorithms using HLS, compatibility issues are expected at this stage
- Partial reconfiguration
    - Do a basic implementation of Partial reconfiguration with dummy configurations

<u>Milestone 4</u>

- Network stack
    - Add a UDP parser to the design
    - Design a custom protocol over UDP for streaming data and register access
- Encryption cores
    - Compatibility issues from HLS implementation of encryption algorithms are to be solved
    - Finish testing
- Partial reconfiguration
    - Attempt an implementation of Partial reconfiguration with available encryption code

<u>Milestone 5</u>

- Connect network stack to encryption cores with and without partial reconfiguration
- Verify the correctness of design without partial reconfiguration
- Detect possible bugs and required modifications for making the design work correctly with partial reconfiguration

<u>Milestone 6</u>

- Implement the required modifications detected in the previous milestone

- Optimize the overall design
- Design a demo to present our project

3.2 Actual Weekly Accomplished Milestones

Milestone 1

- Researched various methods for implementing the network stack using HLS.
- Done tutorials on Vivado HLS and created an example IP block in with the software. Some issues with the environment setup were fixed. Some C libraries for encryption were studied before moving onto the creation of the encryption cores
- Done research on partial reconfiguration

Milestone 2

- Brought up an Ethernet loopback using Xilinx's TEMAC IP-core.
- Encryption algorithms from OpenSSL were tested in Windows. It was functional, but difficult to be implemented in Vivado HLS due to the dependencies.
- Followed a tutorial on partial reconfiguration. Ran into multiple errors as the tutorial was not supported on the Nexys 4 DDr board we had.

Milestone 3

- Developed and tested parser, distributor, ICMP processor, and ARP processor cores. For this milestone, we had ICMP and ARP working on the board.
- We started looking for a basic implementation of encryption instead of using a mature library. We did some research and found some resources for AES. However, after doing minimal alterations, the resulting IP block had a low throughput. From the simulation, the rate of encryption was about 500kB/s, optimization in speed was necessary. The block also needed a wrapper for an AXI interface.
- Successfully ran the tutorial. Implemented partial reconfiguration for a simple demo that included two partially reconfigurable regions that had two configurations each to control the LED lights on the board. Ran into timing problems with clock generation and timing zones.

Milestone 4

- Implemented UDP protocol on the board. Data could be streamed to/from the board using UDP packets. The user could access a register map on the FPGA by sending read/write requests to the board.
- DES core was correctly done and integrated with the networking stack, more optimization needed. AES core produced some mistakes in ciphertexts, needed to re-do
- Successfully completed the demo for partial reconfiguration for the mid-project demo. Started working on Internal Configuration Access Port to reconfigure through the FPGA itself. Midway through the milestone, we decided to remove partial reconfiguration from the project and focus on SD card implementation. Researched SD cards with the remaining time.

Milestone 5

- Spotted some bugs in the network stack when the Ethernet frames were shorter than 60 bytes. Identified the root cause of the problem: padding in the MAC layer to reach a minimum of 60 bytes. Modified all network cores to handle short packets.
- DES core reached a throughput of network line rate through unrolling loops. AES functional error solved, had a rate of 11.7MB/s, which was not far from line rate. Wrapper was also added to the AES core
- Implemented PMOD SD IP core using Microblaze for a simple demo. Tested the ability to read and write to an SD card.

Milestone 6

- Finished the integration of both DES and AES cores into the project.
- Both encryption cores matching with line rate and integrated to the system
- Implemented the AXI DMA to work with the SD card so that it can work with the rest of the project. Integrated SD card functionality to the rest of the project.

4. Description of Blocks

4.1 Network Stack Cores

The overall functionality of the network stack was discussed in Section 1. In this section we review how each block works in more detail.

Parser

This core parses the headers and tries to detect the type the packet received. The first header that can be checked is the Ethernet header which contains an EtherType field specifying the higher level protocol carried by the frame. Parser supports ARP and IPv4 for the EtherType. If this field indicates an ARP packet, the parser immediately sends the packet type message to the downstream core. However, if the frame contains an IP packet, more headers need to be investigated before the packet type is specified. If an IP packet is received, the parser reads the IP header to extract the Protocol field. This field can have many different values, but we only support ICMP and UDP protocols in this project. Again, if an ICMP packet is detected, a message is passed to the next core, but if an UDP packet is received, the parser waits for the UDP header to figure out if the UDP packet is a register access packet or a streaming packet. Once the destination port in the UDP header is read, the parser can distinguish register access requests from streaming data and send an appropriate message to the downstream core.

If the received packet does not fall into any of the categories described above, the parser flags the packet as an unsupported packet and informs the downstream core accordingly. The parser forwards the received packet to a master AXI-Stream interface which is connected to a FIFO buffer.

Distributor

This core waits for a message from the parser. Once the message is received, the distributor reads the packet from the FIFO buffer and routes it to an appropriate protocol processor. If the packet is flagged as an unsupported protocol, the distributor simply discards the packet.

ARP Processor

The Microblaze processor can access this core through an AXI-Lite interface. The processor writes into a few registers containing the board's IP and MAC addresses. The ARP processor waits to receive ARP packets from the distributor and compares the target IP address in the received packet with the board's IP address. If the addresses match, an ARP response packets is generated that includes the board's MAC address.

ICMP Processor

The ICMP processor receives ICMP (ping) packets and compares the destination IP address against the board's IP address. In case they match, it generates an ICMP ping response packet and calculates the ICMP checksum as well. The processor can inform this core about the board's IP address through an AXI-Lite interface.

UDP Depacketizer

This core receives streaming UDP packets and discards all the headers and passes the UDP payload to the downstream cores.

UDP Packetizer

Its functionality exactly reverses what depacketizer does. Receives data from upstream cores and adds Ethernet, IP, and UDP headers to the received data. It also calculates IP header and UDP checksums.

UDP Register Access

This core receives register access packets and initiates memory mapped AXI transactions accordingly. For write requests, no response packets are generated, but for read requests the generated response includes both the address and data.

Aggregator

This core receives the output streams from all network cores and interleaves them using a round robin algorithm to create a single output stream. The aggregator is implemented using an AXI-Stream Switch with multiple slave interfaces and a single master interface.

Tri-Mode Ethernet MAC (TEMAC)

This core, which is available in Vivado's IP catalog, implements the physical and MAC layers. Its registers can be accessed through an AXI-Lite interface from the Microblaze processor. A detailed description of the core's functionality and its register map is provided in its product guide (PG051).

4.2 AES and DES Encryption Cores

AES Encryption Core

Origin of the codes:

https://github.com/mingfengwuye/HLS-AES

In this GitHub link, we only made use of the "AES_VLSI/HLS" directory. There are two source files, namely AES.cpp and AES.h. It is a basic version of C++ code that only encrypts 500kB/s after HLS. In this source, the key was already expanded. We needed to search for the expansion algorithm ourselves and implement it in simulations and in Microblaze.

DES Encryption Core

Origin of the codes:

https://github.com/tarequeh/DES

The baseline software implementation of the DES algorithm used dynamic memory allocation, which is not synthesizable. We replaced all dynamically allocated arrays with constant sized arrays to make the code synthesizable. The initial performance we got from the core was less than 1MB/s. To improve the performance, we replaced loops with high iteration numbers with two nested loops. This coding style allowed us to unroll the inner loop while pipelining the outer one. Partial unrolling of some loops helped us to strike a balance between area and performance.

For both encryption cores, the main modifications done were to pass the compiler in Vivado HLS, to increase speed of encryption, and to provide wrappers to the cores.

Some ways to do the optimization were:

- Re-writing loops into nested loops
- Merging successive loops with the same number of iterations
- Partitioning arrays to enable parallel data access
- Unrolling loops that are the bottlenecks

To write the wrappers, directives were also used in Vivado HLS. We could choose to use AXI Lite for the port for the key and AXI Stream for the input port for the data and output port for the ciphertext.

<u>Testbenches</u>

Testbenches were used in Vivado HLS to verify the correctness and speed of the encryption. Main changes to the testbench was to include the key expansion algorithm so that key is used as the input instead of the expanded key. The testbench was also written to accept a bigger-than-block-size binary file as plaintext input and the output is written to another binary file, as it would be in our hardware applications. The ciphertext would then be compared against the one generated using online resources to verify the correctness. The utilization and speed would be shown as simulation results, where we can understand which loops were the bottlenecks and can focus on optimizing the speed of those loops.

4.3 SD Card

<u>Pmod SD IP block</u>

The IP block that handles interfacing with the SD card. Connects to external SD pins and is an AXI master for the interconnect. Allows reading from and writing to the SD card using Microblaze. Uses the File Allocation Table (FAT) 32 file system to interact with the SD card. SD card testing is done by writing and reading files on the SD card using the example program that comes alongside the IP core. Taken from Vivado library at:
https://github.com/Digilent/vivado-library/tree/master/ip/Pmods/PmodSD_v1_0

<u>AXI DMA</u>

IP block for direct communication with the memory. Is both a slave and a master for the AXI interconnect. M_AXIS_MM2S connected to its own S_AXIS_S2MM. AXI DMA is used to turn the data read from the SD card in blocks to an AXI stream, which can be used by the encryption cores when needed. It is controlled by Microblaze. Taken from Vivado's own IP catalog.

4.4 Others

Microblaze

Microblaze is used to provide control signals to the various IP blocks in our project. Runs the instructions for the overall software that controls data flow, written in C. Taken from Vivado's own IP catalog.

MUX and DEMUX

Demux cores route its input stream to one of its output streams. Mux functions the other way around, routing only one of its input streams to its single out stream. These cores are implemented using AXI-Stream Switch IP core from Vivado's IP Catalog. Their routing functionality is controlled through an AXI-Lite interface controlled by the Microblaze processor.

Mux and Demux are used as selectors that can choose whether the input data for the encryption cores comes from the network or the SD card, whether this data goes into and comes out of the AES or DES encryption core, and whether the output is put into the SD card or transferred over the network.

5. Description of the Design Tree

The Github repository for this project is available at the following address:
https://github.com/soheilshahrouz/G7_Network_Attached_Encryption_Accelerators

The Github repository does not contain the Vivado project, but it includes scripts that allow you to build the project. To build the project, first, you must initialize submodules used in this repository:

```
git submodule update --init --recursive
```

Then, you need to build all HLC cores. Change the directory to /HLS. There is a separate folder for each core. Go to each folder and run the following command:

```
vivado_hls build.tcl
```

Once all HLs cores built, open Vivado in the project's directory and enter the following command in the Tcl console:

```
source ./ECE532_project.tcl
```

When the Vivado project is created, you can generate the bitstream and program the FPGA. However, since the project contains a Microblaze processor, you need to create an SDK project and compile a C++ program as well. To do so, clock on the File->Export->Export Hardware. When a new window is opened, check Include the bitstream and export the hardware. Then, launch the SDK through File->Launch SDK and create an application project. Finally, add the main.cc file in /src folder to your project and build the software project. The generated elf file can be used to program the FPGA. You can also modify the board's network parameters such as IP address and MAC address by changing some constants.

The Github repository contains the following directories:

/src: All HDL files used in the Vivado project along with a c++ source file that is used in the SDK project.

/HLS: All source codes and scripts required for building HLS IP cores.

/IP: .xci files for IP cores that are instantiated in our RTL codes.

/vivado-library: Digilent's IP cores for Vivado. This folder is actually a submodule.

/ECE532_project: Once the project is built, this directory contains the Vivado project.

/docs: Group report and slides for the final demo.

6. Tips and Tricks

- Planning is very important for a large project with multiple parts. There should always be a plan for when things go wrong and don't work out. Always have backups and plan for failure, both in overall planning and working on the project itself.

- Communication between members is very important. Don’t try to do everything yourself if one of your teammates can help with their knowledge. Teamwork is very important.
- There are a lot of open source resources online available to use. You should always spend decent time researching if something exists already before trying to build it yourself.