Implementing Network Stack on a FPGA

Aaditya Kothary
University of Illinois Urbana-Champaign
Champaign, IL

Parithimaal Karmehan
University of Illinois Urbana-Champaign
Champaign, IL

ABSTRACT

FPGAs are experiencing a renewed prevalence in Communication Networks today [9]. Due to their unparalleled flexibility compared to ASICs and efficiency compared to CPUs, they are being commonly used to build network cards, and offload packet sorting capabilities to hardware. This lowers latency, power consumption and area, while leaving room for reprogramming and improvement [6]. As sophomores taking classes in Digital Design and Communication Networks, we wanted to try our hand at applying the two concepts together, and in doing so gain insights into this field. We tried our hand at adding network capabilities to the Urbana FPGA Board [13], starting from the physical layer up till the application layer. We were ultimately successful, becoming the first group to connect the Urbana FPGA Board to the internet.

ACM Reference Format:
Aaditya Kothary and Parithimaal Karmehan. 2024. Implementing Network Stack on a FPGA. In *Proceedings of ACM Conference (ECE438).* ACM, New York, NY, USA, 6 pages. https://doi.org/

1 INTRODUCTION

1.1 Our Motivations

When we set out to complete this project, one of the biggest things we thought about was how we could leverage our knowledge of ECE along with the understanding of networking concepts from this class to implement a project that would serve as a capstone to our learning in this class. We knew that we wanted to build something that involved both software and hardware. Eventually we decided on building a full network stack on an FPGA that would involve lower layers being implemented in hardware and upper layers of the Network stack being implemented in Software. This project was one that both of us enjoyed as there were many bugs that we had to find which gave us a deeper understanding of the different layers in the network stack and our codebase. Barring the toolchain issues that we had with AMD Vivado and Vitis IDE, we truly learnt a lot and pushed the boundaries of our knowledge with this project.

This was fueled by our passion for embedded systems and real-time systems. With the advent of the Internet of Things and small embedded systems such as smartwatches, smart glasses etc, the question we had was how quickly can we communicate with other devices on the Internet [8]. By using an FPGA, we were able to leverage its strengths in latency and determinism to build a fast, cycle accurate machine. Furthermore, an FPGA allowed us to implement the Physical and Link layers in hardware whilst we were able to implement the Network, Transport, and Application layers in software on a soft processor.

A common use of FPGAs in networking are in High-Frequency Trading (HFT) applications which build Network Stacks that utilise UDP for the transport layer to accelerate the speeds of communication and retrieve market data to make decisions in split seconds [9]. Whilst we were inspired by this, we wanted to build something slightly more complex which made use of the application layer as well.

1.2 The Approach

Our final goal was to connect the Urbana Board straight to the Internet over a Ethernet cable to an IllinoisNet endpoint. We chose to build our layer using a RAW API that uses Callback functions instead of a Sockets based API. This enabled us to make a 'bare-metal' network stack that is more lightweight and low-power compared to a network stack implemented on FreeRTOS or PetaLinux [11].

We built the Hardware layers first using open source Intellectual Property (IP) blocks and SystemVerilog code to synthesize the lookup tables on the FPGA. We then tested this by looking at the low level signals for a hardcoded ethernet frame on an oscilloscope. Verifying that the data being transmitted made some sense and that the hardware was working fine, we decided to continue further with building software layers on top.

We first ran a TCP server/client on the FPGA and connected it to our laptops through a link layer switch, as per the image below. We ran iperf and verified that this code was working. Following that we built a http client based on the TCP client and tested that by running a http server on our laptop and making a GET request to retrieve files from the server. Once this was working, we added functionality to resolve DNS queries and return an IP address for a given domain name. Once we built this, we connected to IllinoisNet and made queries to HTTP sites to test our code.

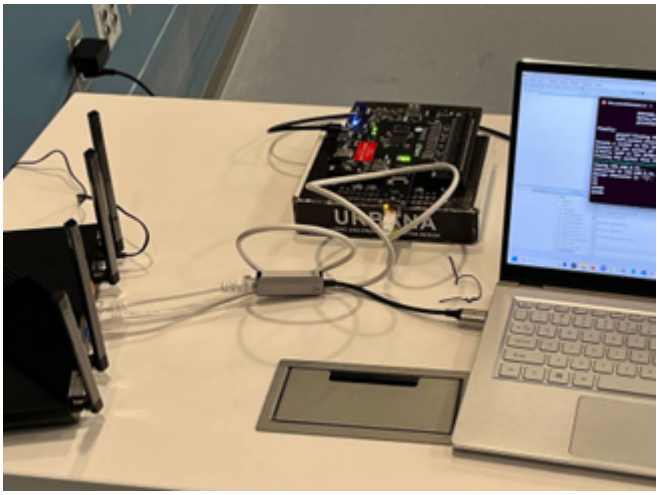

Figure 1: LAN set up for testing TCP/IP

Figure 2: Image of the LAN8720

2 RELATED WORK

While Networking is an often mentioned project idea in FPGA forums, not many have followed through to implementation. Among the completed projects, there is a lot of variation due to the different types of FPGAs on offer. Even amongst Xilinx FPGAs, there is a huge amount of variation, as different boards have different types of peripherals, and come with varying levels of Board Support Packages. Most of the documented Xilinx projects with Networking have been done on Zync-7000 SOC Boards, or the Artix-7 series Boards. Both of these come with an RJ45 ethernet port already soldered onto the board with preset pinouts, and have Board Support Packages for harnessing the external memory present on the Boards (such as DDR3) with relative ease.

The Urbana Board, on the other hand, does not have its own Ethernet Port, or PHY [13]. This meant that we had to obtain external hardware and deduce how to connect it correctly to the Urbana Board. The Urbana Board also did not come with a Memory Interface customized for using DDR3, meaning we had to read through Xilinx's memory interface documentation in order to customize Vivado's Memory Interface IP core for our needs [12].

For the Network and Transport Layers, we planned to harness lightweight IP (lwIP), an open source TCP/IP stack built by Adam Dunkels that is intended for use in embedded systems and low power devices, including FPGAs [7]. There is slightly more documentation for lwIP, as the implementation is more Board-agnostic, and Xilinx itself has some example lwIP code in its template applications [10]. However, as we will cover in the next section, there were several bugs in the example code provided by Xilinx, and all lwIP Xilinx projects online focused on just setting simple TCP servers or clients.

3 SYSTEM DESIGN

3.1 Physical Layer

As mentioned previously, the Urbana Board did not come with its own RJ45 ethernet port, hence we had to purchase external hardware. We settled on the LAN8720, which had an ethernet port and a PHY module. The Phy uses a LAN8720 Application-Specific Integrated Circuit (ASIC) to transceive data between the physical medium of the Ethernet Cable and the digital logic that we did our higher level processing on [16]. The LAN8720 PHY signals are then broken out into PMOD pins, which we used to connect directly into the PMOD ports on our Urbana Board. These signals include the TX/RX, MDIO, Power and clock signals, which will then communicate with the MAC block. The MAC, as well as remaining layers are housed directly inside of the FPGA.

3.2 Link Layer

As the LAN8720 PHY communicates with a Medium Access Control (MAC) block using RMII signals, while the FPGA communicates using MII signals, we needed a signal converter in order to interface between the FPGA Board and the LAN8720. Sourcing an appropriate IP required extensive research, but we eventually found an open source MII to RMII IP that was published by Analog Devices [14]. Using the provided MakeFiles and Vivado's TCL console to build the converter, we were able to generate an MII to RMII converter that we could then use in our block design.

Our project needed a significant amount of cacheable memory in order to store ethernet frames and packets in the Network/Transport Layers. This memory would also become

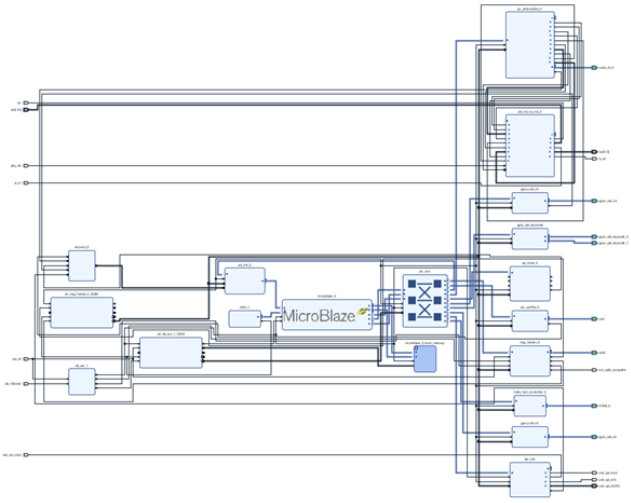

Figure 3: Vivado Block Design

necessary later down the line when setting up DNS caches in our application layer. Hence, we had to customize Vivado's Memory Interface Generator IP to generate the logic required for the processor to communicate with onboard DDR3 SDRAM IS43TR16640CL-125JBL [5]. This involved figuring out memory related timing constraints such as the four address width, row to column delay, row to address delay among others [12]. Furthermore, there were more than 20 individual signals for DDR3 SDRAM that ended up with really long paths for the which lead to the synthesis and path delay between lookup tables on the FPGA being long. This was one of the constraints that lead us to have slower timing than expected. This is also one of the shortcomings of using premade development boards for such projects where timing constraints are extremely important.

The SDRAM provided us with sufficient memory for Transport and Network layer buffers and small DNS caches for more efficent lookup of domain names. The final hardware block design that is the source of our hardware bitstream is included in Figure

3.3 Network Layer

The network layer was the first point where we implemented functions in software using lwIP. Even though we had open source code to base our software code on [7], the network layer was where we had to interface with the hardware on the Urbana board, which involved modifying the lwIP settings in Vivado for our board.

Our "hello world" analogue was essentially setting up a TCP echo server that sends back to the client any data that it sent to the server. While Xilinx provides an implemention of this using lwIP in Vitis [10], there are several bugs that prevent the example from even running correctly.

Firstly, there is a bug in xadapter.c when setting populating the lwIP data structure for type of TEMAC used. Due to two missing break statements, the program always default to an error, stating that it is unable to determine the TEMAC type used, even if it was able to configure the data structure correctly [15].

Another bug was that Vitis would not detect the AxiEthernetLite IP in our block design as a valid MAC block, and would prompt the user to create a MAC block even though one already existed in hardware. Thanks to some patches shared on a Xilinx Forum, we were able to resolve this issue relatively quickly [3].

Another undocumented issue, not so much as a bug, was that lwIP is by default set to autonegotiate the link speed on the ethernet wire. This works only as long as the ethernet port on the opposing end of the ethernet wire was also configured to autonegotiate the link speed instead of defaulting to a specific link speed. Otherwise, the autonegotiation would fail, and the program would end prematurely [2].

lwIP allows for dynamically assigned IPs, and only defaults to a preset one in the absence of a DHCP server nearby. To run the echo server example, we had to set up a Home Router, which connects to the FPGA and a laptop using ethernet wires mapped to separate ports on the router.

3.4 Transport Layer

After establishing the echo server, we then proceeded to test the TCP client and server examples. This involved largely reusing the code base from the echo server example, but configuring the server to respond to iPerf commands, in order to test the bandwidth when running the code as both client and server [1].

One debugging hurdle we had to overcome was that we were able to connect to a TCP server on the board while running TCP client in Windows Powershell on our board, but not when running the client inside of Windows Subsystem for Linux (WSL). We eventually deduced that this was because WSL2 has its own IP address that is separate from the PC, and is not advertised to other devices in the PC's local network. In order for an external TCP client to connect to a TCP server running inside WSL, we had to implement port forwarding from a port on the Windows Laptop to a port for the Ubuntu shell running inside Windows.

Our main means of debugging at this step was by running Wireshark on the ethernet adapter of our PC, and sniffing on incoming packets from the FPGA via the router. This allowed us to determine the order of packets sent when the FPGA started up, whether the sent packets were even arriving at their intended destination, and whether our laptop was

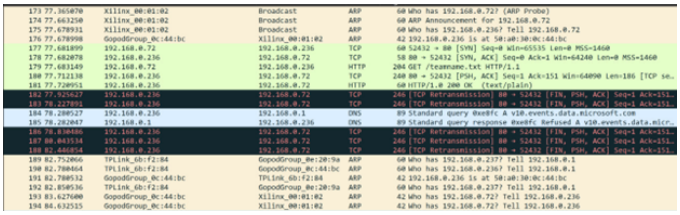

Figure 4: Wireshark capture of TCP echo server

responding in kind with its own packets. One screenshot of our wireshark captures is included as Figure 4, which depicts the FPGA making a HTTP request to a server running on a laptop on the same LAN as the FPGA.

3.5 Application Layer

The next step for us after establishing the transport layer was to build an application layer on top of it so that our FPGA can retrieve and display data from network servers on the edge of the Internet. To do so, we had to build support for 3 different protocols.

The first protocol we focused on was the Dynamic Host Configuration Protocol (DHCP). Most of the implementation for this was already done for us based on the TCP client base code that we had used for the transport layer. As such, to implement this in software, all we had to do was to set some macros `LWIP_DHCP = 1` and use the `dhcp_start(struct netif *netif)` function in our main code to make a DHCP request and obtain an IP address.

The second protocol we implemented was HyperText Transfer Protocol (HTTP). To do this, we used more lwIP code generate a HTTP request string and encapsulate that message in a TCP packet to send it over the TCP protocol. Whilst writing the C code was pretty simple and involved using lwIP library functions such as `httpc_get_file_dns`, we had to make sufficient edits to our makefile to compile our project successfully. We had to add new file lists and expand the VPATH of our makefile to ensure that the correct c files were linked.

Lastly, we implemented Domain Name Service (DNS) alongside HTTP to make requests based on a domain name instead of a specific IP address. This meant implementing a small (four entry large) cache of previously looked up results as we were not expecting to make requests to many websites. When a DNS query is made, we would check if the resolution to the domain name exists in the cache. If not, we would enqueue a DNS query to our ISPs' DNS servers which could be from IllinoisNet, Pavlov Media or Volo depending on where we were testing. Testing DNS resolution required switching from our earlier test set ups, switching the LAN's connection from LAN to a WAN. This involved connecting the FPGA

Figure 5: WAN Set up

directly to a wall ethernet port in one of the University buildings, as per Figure 5.

Once we developed all these capabilities, we tested our final project on three main websites:

(1) example.com
This website provided the most simple proof-of-concept that we were able to get HTML data from files in example.com such as index.html
(2) api.nbp.pl
This website allowed us to retrieve real-time currency data from a Polish Bank sendback JSON data that we would be able to parse at our application layer

Figure 6: Serial Port Output of a GET request to api.football-data.org

(3) api.football-data.org
This API provides real time data on football matches, such as league standings, upcoming schedules, and historical information. Their database includes the top-5 leagues in Europe, as well as competitions such as Champions League and Europa League, and statistics on several popular football players. An example output is encapsulated in Figure 6.

4 EVALUATION

Our project was a general success in that we achieved our goal of connecting the Urbana Board to the internet, and being the first to do so in the process. We learned a lot about how network concepts taught in CS438 are actually applied in the real world, and how different layers interface with one another. For example, we learned that our laptops are not automatically configured to run as switches, hence it is easier to connect the Board to a Home Router and then connecting it to multiple devices, for thorough testing and debugging.

There were some topics that we ended up learning through trial and error, which were covered in the latter weeks of CS438. One insight was the MAC address is not given to a Computer per se, but assigned to the ethernet adapter or WiFi adapter that is connected to the Computer. Another insight was that an ARP broadcast is sent before any TCP packets are sent, to determine where an IP address is located in the network, by linking it with the appropriate MAC address.

We also learned a great deal about how IllinoisNet functions as a whole, as we spent a great deal of time debugging using an ethernet port in ECEB 3026 (ECE391 lab), only to find out that the ports in that room blacklist MAC addresses that do not belong to any of the lab computers. Thanks to the kind assistance of Engineering IT Associate Specialist Jonathan Kim, we were able to find out which ethernet ports in the ECEB and Siebel were patched in, and could be used for our project.

Nevertheless, there are several avenues for improvement on our project. The first would be to implement a Transport Layer Security layer into lwIP on top of TCP, that would enable us to interface with HTTPS APIs, and not just HTTP. This would be a significant expansion as most authentication based APIs used TLS for added security, and many organizations even mandate it in order to interface with their products [4].

Another area for improvement would be to offload the network and transport layers onto hardware, as that is where FPGAs are most commonly used in industries such as HFT [9]. We did not pursue this route initially as it would be a tall order for a first time networking project, and we wanted to get a proof-of-concept working on this board before optimizing the network stack on the board.

Lastly, for a more seamless user experience, the project could be expanded to interface with a HDMI display to print packet outputs, allowing for more seamless user interaction. As it stands, our project only allows for user input via a USB Keyboard, to input the domain name and URI for forming a HTTP GET request, before displaying the results within a serial output within Vitis itself.

5 CONCLUSION

To conclude, this project was a great way for us to dip our feet into the world of digital design and networking. We now have a greater appreciation for the networking hardware that many of us take for granted, as well as the software protocols that work tirelessly within our devices to ensure that we are able to stay connected to one another in real time.

We sincerely hope that future students will feel similarly inspired to make FPGA-based networking applications, and build off of our work to arrive at more impressive results!

REFERENCES

[1] [n. d.]. iPerf - iPerf3 and iPerf2 user documentation. ([n. d.]). https://iperf.fr/

[2] [n. d.]. WHAT IS ETHERNET AUTO-NEGOTIATION? ([n. d.]). https://www.fiberoptics4sale.com/blogs/archive-posts/95041222-what-is-ethernet-auto-negotiation#:~:text=Auto%2Dnegotiation%20is%20a%20protocol,configuration%20of%20the%20other%20side.

[3] 2021. AXI EthernetLite Vitis errors with lwIP: "Failed to create application project" + No Ethernet MAC IP instance in the hardware. (2021). https://support.xilinx.com/s/question/0D52E00006hpQR2SAM/axi-ethernetlite-vitis-errors-with-lwip-failed-to-create-application-project-no-ethernet-mac-ip-instance-in-the-hardware?language=en_US

[4] 2023. Importance of SSL in a company. (2023). https://primasecure.com/importance-of-ssl-in-a-company/#:~:text=Without%20SSL%20certificates%2C%20businesses%20are,of%20their%20online%20security%20strategy.

[5] 2023. IS 43 Series SDRAM Datasheet. (2023). https://www.issi.com/WW/pdf/43-46TR16640C-81280CL.pdf

[6] Pabudi T Abeyrathne, S. Devapriya Dewasurendra, and Dhammika Elkaduwa. 2021. High Frequency Trading Acceleration Using FPGAs. *2021 IEEE 30th International Symposium on Industrial Electronics (ISIE)* (2021).

[7] Adam Dunkels. 2002. lightweight IP documentation. (2002). https://www.nongnu.org/lwip/2_1_x/index.html

[8] Mohammed Elnawawy, Abid Farhan, Ahmad Al Nabulsi, A.R. Al-Ali, and Assim Sagahyroon. 2019. Role of FPGA in Internet of Things Applications. (2019). https://www.researchgate.net/publication/339403589_Role_of_FPGA_in_Internet_of_Things_Applications

[9] Christian Leber, Benjamin Geib, and Heiner Litz. 2011. High Frequency Trading Acceleration Using FPGAs. *2011 21st International Conference on Field Programmable Logic and Applications* (2011).

[10] Anirudha Sarangi, Stephen MacMahon, and Upender Cherukupaly. 2014. lightweight IP documentation. (2014). https://docs.amd.com/v/u/en-US/xapp1026

[11] Unknown. 2017. The difference between the standalone and linux application project. (2017). https://support.xilinx.com/s/question/0D52E00006hpdLcSAI/the-difference-between-the-standalone-and-linux-application-project?language=en_US

[12] Unknown. 2018. DS176 - Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v4.2 Data Sheet (DS176) (v4.2). (2018). https://www.xilinx.com/products/intellectual-property/mig.html#overview

[13] Unknown. 2021. Urbana Board. (2021). https://www.realdigital.org/hardware/urbana

[14] Unknown. 2022. UTIL MII TO RMII. (2022). https://wiki.analog.com/resources/fpga/docs/util_mii_to_rmii

[15] Unknown. 2023. Is this a bug – case without break? (2023). https://support.xilinx.com/s/question/0D54U00006jQBK5SAO/is-this-a-bug-case-without-break?language=en_US

[16] Unknown. 2024. LAN8720 documentation. (2024). https://ww1.microchip.com/downloads/aemDocuments/documents/UNG/ProductDocuments/DataSheets/LAN8720A-LAN8720Ai-Data-Sheet-DS00002165.pdf