ECE1373S Project Report

FPGA Shell

Nariman
Omar
Ming

Department of Electrical and Computer Engineering
University of Toronto
June 23, 2017

Contents

List of Figures

List of Tables

1 Introduction

Applications which need datacenters are increasing rapidly; however, the datacenter improvements have slowed considerably because of "power wall". In this regard, using specialized system could help users increase the performance and lower the power consumption. ASIC technology is fully specialized that could improve both power efficiency and performance. However, they do not have any flexibility, so, this prevents them from finding their way into datacenters. The more flexible alternative for ASIC is FPGA. FPGAs are more efficient in comparison with general purpose processors in terms of performance and power. The problem with this component is that using them adds quite a few efforts on building the desire system. Furthermore, managing this resource alongside other resources in datacenter is a tough task. One of the biggest problem is the interconnections through which these resources are going to communicate.

In this report, we are going to introduce a "shell", which has all the interconnections that a user may need such as Network (Ethernet, UDP, and TCP), PCIe, and DDR memory. Thus, the users only need to design their application with a simple input and output, then it could be integrated into the shell in a convenient manner, and the user does not need to deal with interfacing. As an application for this shell, we have implemented the LDPC decoder algorithm min-sum in HLS. The min-sum decoder is a graphical inference algorithm used to decode a message transmitted through a noisy channel. Some of the benefits of this decoder is that its major computations are simple and they can be easily parallelized. This make the decoder ideal for hardware implementation on FPGA and a suitable application to test our system.

2 Current Status

Table 1 has all the information about the current status of the system. the following items could be added in the future:

- Make the application work with UDP interface.
- Make a generic script for shell generation regarding the user specifications.
- Add partial reconfiguration flow to the generator script.

3 Initial Architectural Design

In 2014, Microsoft deployed FPGAs in their data centers in order to accelerate the bing search engine. Their work demonstrated that they could double the performance with a 10% overhead of power and cost [1]. The large scale deployment of FPGA in datacenters, proved that using FPGAs in datacenter is useful, because these state-of-the-art components provide higher performance with lower power consumption. Nevertheless, there are many reasons which cause the fact that FPGAs are not widely used in systems. One of which is the FPGAs should be programmed for special purpose application, and design effort is so much more than develop a code for general purpose computation resources. The other reason is that the data is usually hosted in a memory which is tightly coupled with a CPU, and moving this data to FPGA would be considered as an overhead.

	HLS test	System test	System integration	Comments
UDP	✓	✓	✓	The UDP Loopback is tested using a python script.
TCP	✓	✓	✓	The TCP echo application is tested. there is something wrong with transmitting large amount of data.
PCIe	✓	✓	✓	It is tested using a C code, which writes data to the memory, and read it back and do the comparison.
App:min-sum stream	See comment	✗	✗	Using HLS testbench with streaming interface works but when wrapping the module to work with UDP, it doesn't work with the overall system.
App:min-sum AXI	✓	✓	✓	Using PCIe data is written to DDR memory. The module reads data, and writes the results into DDR memory. The host read data through PCIe, and compare it with Matlab generated golden model.
DDR4 memory	✓	✓	✓	Used PCIe test
Ethernet	✓	✓	✓	Tested Using Ping command

Table 1: Status of shell modules

In addition, implementing high speed links for moving data is not very straight forward.

In this project we are going to address the problems with using FPGAs. "FPGA Shell" is the term that we have chosen for our project. Shell is going to have all the interconnections a user might need. So the user does not need to deal with all the details of interconnection, and he or she just needs to put their application into the shell. The biggest inspiration of this project is catapult shell that you can see in Figure 1. Initially we decided to have one application per FPGA, so we did not need to have a router inside the FPGA, and we decided to have DDR memory, PCIe, DMA engine, Ethernet, and higher layer of network like UDP and TCP. So, we ended up with shell that is illustrated in Figure 2.

High-Level synthesis (HLS) methodologies have emerged to reduce the FPGA design time. As a result, more designer can enjoy this technology [2], [3]. Using HLS, Xilinx has a open source project [4] that has entire TCP/IP stack in it. However, this code was pure Verilog and it was hard to debug. So, during porting that project for Alpha Data 8k5 [5], we have implemented everything in block Diagram for debug purposes and to be more user friendly.

For testing this platform, we decided to implement an application which is described in detail in next section. The original interfacing for the application regime was a single AXI-Stream input and a single AXI-Stream output. The user will then define his/her data packet according to their application design. The application's AXI-Stream I/O will then be connected to the UDP core to send data back and forth between the host computer and application.

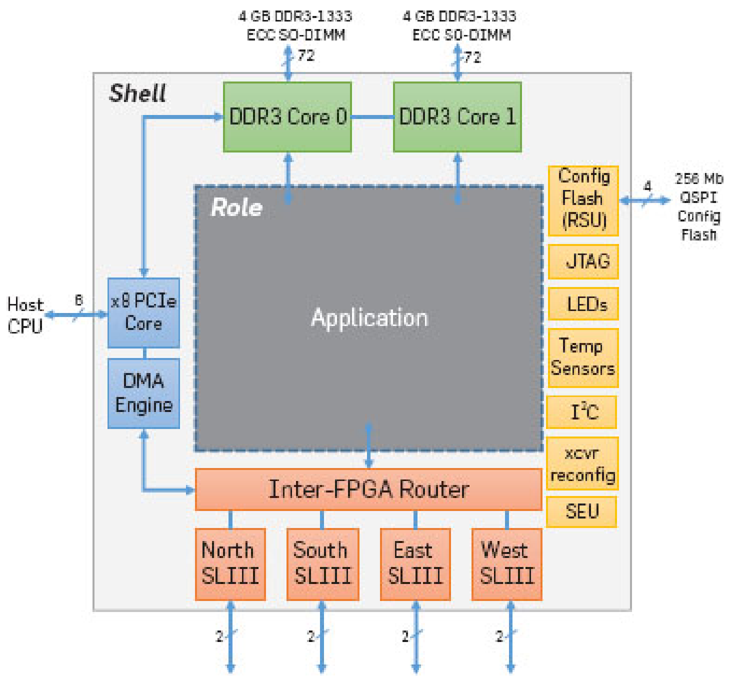

Figure 1: Microsoft Catapult Shell block diagram

4 Application: Min-sum decoder

Low-density parity-check (LDPC) codes are a class of linear block error control codes used for correcting digital data transmitted over an unreliable channel. They were discovered by Gallager in 1962 [6] and then were rediscovered by Mackay to achieve near Shannon limit performance in 1993 [7]. The LDPC decoder uses a message-passing algorithm on a graph know as belief propagation to correct potential errors in the transmitted message over a noisy channel. A simpler approximation to belief propagation is the min-sum algorithm. Min-sum reduces the computation and hardware implementation complexity of belief propagation while maintaining comparable performance [8].

LDPC codes have become increasingly popular in the past years and are currently being used in multiple digital communication standards such as satellite (DVB-S2 and DVB-T2), wired (IEEE 802.3 10 GBASE-T), and wireless (IEEE 802.11n/ac/ad) transmissions [9]. This wide industry adoption stems from the LDPC decoder's algorithmic simplicity and its intuitive hardware implementation.

In this section, we will briefly introduce the necessary background to understand the LDPC

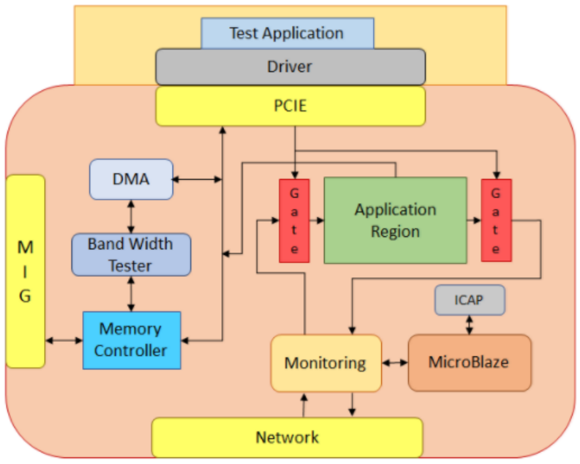

Figure 2: Shell block diagram

decoding algorithm (min-sum) followed by the detailed explanation of the FPGA implementation of the LDPC decoder in HLS.

4.1 Communication Channel Model

For the purposes of this project, we will look at a simple block diagram consisting of an encoder, channel and decoder illustrated in Figure 3.

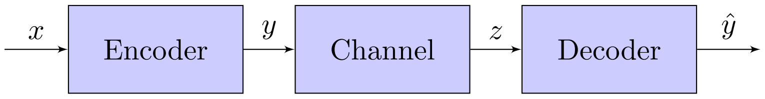

Figure 3: Digital communication channel with an encoder and decoder

The input to the encoder is a block of information bits (zeros and ones) of length k. The encoder will add some redundant bits for the decoder to correct potential errors. The addition of redundant bits is a simple linear map using a binary matrix know as the generator matrix G. Therefore the binary encoding is described as

$$v = x \cdot G, \tag{1}$$

where the sizes of v, x, and G are $1 \times n$, $1 \times k$, and $k \times n$ respectively. Note that (1) is a binary or *mod* 2 operation and $n > k$. The output of the encoder v is known as a *codeword*.

For example, for the $(7,4)$ Hamming code where $k = 4$ and $n = 7$, the generator matrix G is

$$G = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}. \tag{2}$$

If $x = \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix}$, then the encoding is

$$v = x \cdot G = \begin{pmatrix}1 & 0 & 1 & 1\end{pmatrix} \cdot \begin{pmatrix}1 & 1 & 1 & 0 & 0 & 0 & 0\\ 1 & 0 & 0 & 1 & 1 & 0 & 0\\ 0 & 1 & 0 & 1 & 0 & 1 & 0\\ 1 & 1 & 0 & 1 & 0 & 0 & 1\end{pmatrix} = \begin{pmatrix}0 & 1 & 1 & 0 & 0 & 1 & 1\end{pmatrix} \tag{3}$$

The next step is to map the binary signal to $+1$ and -1 which is know as non-return-to-zero (NRZ) line coding. The mapping function is

$$y_i = \begin{cases} -1 & \text{if } v_i = 0 \\ +1 & \text{if } v_i = 1 \end{cases} \quad \text{for } i = 1, \ldots, n. \tag{4}$$

The channel model is an additive white Gaussian noise (AWGN) illustrated in Figure 4. Each element of the vector w is independent and identically distributed according to a Gaussian distribution with zero mean and variance σ^2, i.e. $w_i \sim \mathcal{N}(0, \sigma^2)$.

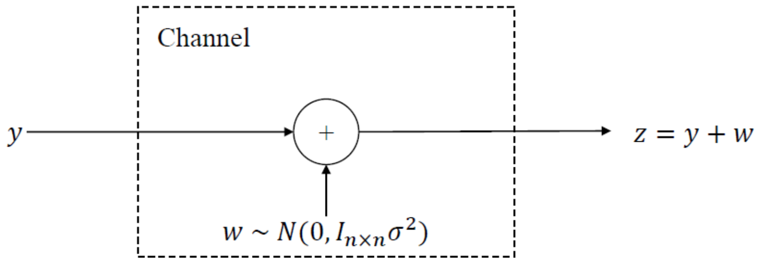

Figure 4: Additive white Gaussian noise channel

When the decoder receives the output of the channels, it calculates the log-likelihood ratio (LLR) as

$$LLR_i = \ln\left(\frac{\Pr(z_i = +1 \mid y_i)}{\Pr(z_i = -1 \mid y_i)}\right) \quad \text{for } i = \{1, \ldots, n\}, \tag{5}$$

where $\ln(\cdot)$ is the natural logarithm. For a Gaussian distribution, the LLR can be simply calculated as

$$LLR_i = \frac{-2 \cdot z_i}{\sigma^2}. \tag{6}$$

The LLR, as stated in (5), simply indicates whether the transmitted symbol was more likely a $+1$ or -1. If the LLR is positive, it means that the transmitted value has a higher probability of being $+1$ and vice-versa. As we will see later, the LLRs are vital for the decoding algorithm.

4.2 Tanner graph

Another way to describe a code is through the parity-check matrix H. The parity-check matrix is defined as the *left* null-space of the generator matrix G and has a size of $(n-k) \times n$. Therefore, every row of the H is orthogonal to every row in G. For example, the parity-check matrix of the $(7, 4)$ Hamming code is

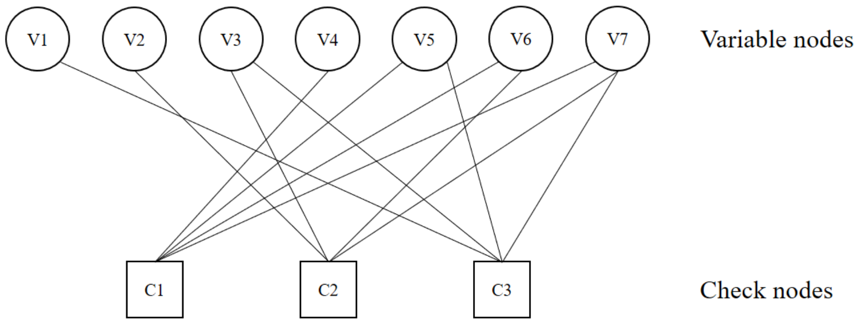

Figure 5: $(7,4)$ Hamming code tanner graph

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}. \tag{7}$$

Notice that the multiplication of $G \cdot H^T$ defined in (2) and (7) is $\mathbf{0}$. Since the codeword v in (1) is a linear combination of the rows G, then $H \cdot v^T = 0$; this is know as the codeword membership test and it is essential to identify if a binary vector of length n is a codeword or not.

Using the parity-check matrix, the code can be represented as a bipartite graph. The first set of vertices are known as variable nodes and the other set are know as check nodes. There are n variable nodes and $n-k$ check nodes corresponding to the number of columns and rows of H respectively. The edges of the graph correspond to a "1" in the parity-check matrix. Figure 5 represents the graph of the $(7,4)$ Hamming code. This is know as a Tanner graph.

4.3 Message passing and min-sum algorithm

The message passing algorithm exploits the graphical structure of the Tanner graph by sending messages along the edges of the graph between the variable and check nodes. These messages are updated in an iterative manner. After each iteration, the decoder uses the message to check whether it has produced a valid codeword. We will introduce some notation to help explain the message passing algorithm.

Let $\mathcal{I} = \{1, 2, \ldots, n\}$ be the set of variable nodes indices and $\mathcal{J} = \{1, 2, \ldots, n-k\}$ be the set of check nodes indices. The neighborhood of check $j \in \mathcal{J}$, denoted as $\mathcal{N}_c(j) \subseteq \mathcal{I}$, is the set of variable node indices connected to check j. Similarly, the neighborhood of variable $i \in \mathcal{I}$, denoted by $\mathcal{N}_v(i) \subseteq \mathcal{J}$, is the set of check node indices connected to variable i. Furthermore, $\mathcal{N}_c(j) \setminus i$ is the neighborhood of check j excluding variable i. For examples, in Figure 5, the neighborhood of check node 2 is $\mathcal{N}_c(2) = \{2, 3, 6, 7\}$ while the neighborhood of variable node 6 is $\mathcal{N}_v(6) = \{1, 2\}$. Let $q_{i \to j}$ denote the message from variable i to check j and $r_{j \to i}$ denote the message from check j to variable i.

The min-sum decoder, fully described in Algorithm 1, aims to correct any errors in the channel output vector z by updating the messages or beliefs in the Tanner graph. The algorithm starts off by calculating the LLRs (denoted by γ in Algorithm 1) according to (5). Then, it sets all of the messages from variable $i \in \mathcal{I}$ to all its check neighbors equal to γ_i. The check-to-variable

Algorithm 1 Min-sum algorithm

```
Require: Noisy output from channel $z$ and channel variance $\sigma^2$
Ensure: Soft output $\hat{y}_s$ and hard output $\hat{y}_h$
 1: Step 1: LLR calculation and message initialization
 2: for $i \in \mathcal{I}$ do
 3:     $\gamma_i \leftarrow -2 \cdot z_i \ / \ \sigma^2$ {LLR initialization as in (5)}
 4:     for $j \in \mathcal{N}_v(i)$ do
 5:         $q_{i\to j} \leftarrow \gamma_i$
 6:     end for
 7: end for
 8: repeat
 9:     Step 2: Check to variable message update
10:     for $j \in \mathcal{J}$ do
11:         for $i \in \mathcal{N}_c(j)$ do
12:             $r_{j\to i} \leftarrow \big(\prod_{i' \in \mathcal{N}_c(j)\backslash i} \mathrm{sign}(q_{i'\to j})\big) \cdot \big(\min_{i' \in \mathcal{N}_c(j)\backslash i}(|q_{i'\to j}|)\big)$
13:         end for
14:     end for
15:     Step 3: Variable to check message update
16:     for $i \in \mathcal{I}$ do
17:         for $j \in \mathcal{N}_v(i)$ do
18:             $q_{i\to j} \leftarrow \gamma_i + \sum_{j' \in \mathcal{N}_v(i)\backslash j} r_{j'\to i}$
19:         end for
20:     end for
21:     Step 4: Calculate soft and hard decisions
22:     for $i \in \mathcal{I}$ do
23:         $\hat{y}_{s,i} \leftarrow \gamma_i + \sum_{j \in \mathcal{N}_v(i)} r_{j\to i}$
24:         $\hat{y}_{h,i} = \begin{cases} 1 & \text{if } \hat{y}_{s,i} < 0 \\ 0 & \text{otherwise} \end{cases}$
25:     end for
26: until $H \cdot \hat{y}_h^T = \mathbf{0}$ OR reached maximum number of iterations
```

and variable-to-check updates can be comprehended better using an example. Let's start with the check-to-variable update. Suppose we want to update the message from check 3 to variable 1, $r_{3\to 1}$, in the example in Figure 5. We only need the other incident messages to check 3 to perform the update: in this example, we consider $q_{3\to 3}$, $q_{5\to 3}$ and $q_{7\to 3}$ and perform the check node calculation as follows:

$$r_{3\to 1} = \big(\mathrm{sign}(q_{3\to 3}) \cdot \mathrm{sign}(q_{5\to 3}) \cdot \mathrm{sign}(q_{7\to 3})\big) \cdot \min\big(|q_{3\to 3}|, |q_{5\to 3}|, |q_{7\to 3}|\big). \tag{8}$$

Figure 6 illustrates the flow of messages in the previous example. For the variable-to-check update, let's consider the message from variable 7 to check 1, $q_{7\to 1}$, in Figure 5. Similar to the previous example, we will consider the other incident messages namely $r_{2\to 7}$ and $r_{3\to 7}$ to perform the variable node calculation as follows:

$$q_{7\to 1} = \gamma_7 + r_{2\to 7} + r_{3\to 7}. \tag{9}$$

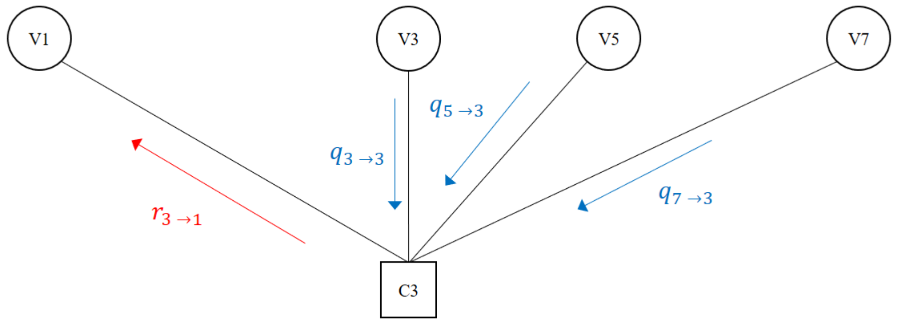

Figure 6: Update $r_{3\to1}$ example

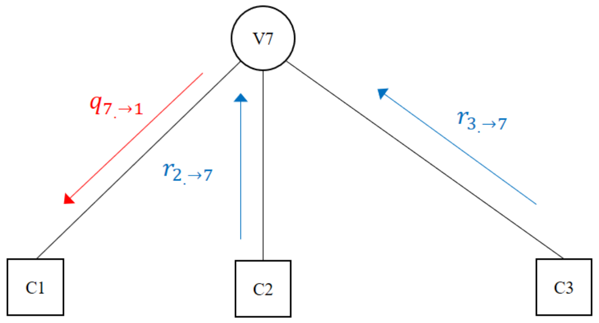

Figure 7: Update $q_{7\to1}$ example

Figure 7 illustrates the flow of messages in the variable-to-check example. Calculating the soft decision after each iteration is done by simply summing all of the incoming message to a variable node along with its initial LLR.

4.4 LDPC and QC-LDPC codes

LDPC codes are a class of linear block codes which are decoded using the min-sum algorithm. As the name suggests, LDPC codes have a sparse parity-check matrix (few ones in H). For *regular* LDPC codes, the number of ones in each column, l, is equal and the number of ones in each row, r, is equal. For example, a $(3, 6)$ regular LDPC has exactly $l = 3$ ones in each column and $r = 6$ ones in each row. From a Tanner graph's perspective, the number edges connected to each variable node, also known node degree, is equal and each check node degree is equal.

In the HLS implementation of the min-sum decoder, a subclass of LDPC codes called Quasi-Cyclic LDPC codes were used. To construct the parity check matrix of a QC-LDPC, H is divided into smaller square blocks of equal size $p \times p$. Each sub-block is simply a $p \times p$ identity matrix shifted to the right by some value. We denoted sub-block by I_s where s is the amount of right shift. For example, (10) shows QC-LDPC parity-check matrix with the sub-block size equal 3.

$$H = \begin{pmatrix} I_0 & I_0 & I_0 \\ I_0 & I_1 & I_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{10}$$

QC-LDPC codes are beneficial when dealing with hardware for two main reasons:

1. We only need to store the shifts of the sub-blocks instead the entire parity-check matrix
2. It allows the variable and check node computations to be partially-parallel instead of fully-parallel. If the block length of the code is very large, a fully-parallel implementation on a FPGA can be infeasible.

5 Specification Evolution and Final Architectural Design

In this project, most of the items, which were introduced in Section 3, are implemented. However, due to the time limitation of this project, we could not add the partial reconfiguration option to the shell. Partial reconfiguration is an important aspect of the shell because it can can reduce synthesis and implementation time and reconfiguration time. The other important aspect of Partial reconfiguration is because of the PCIe; if the PCIe device is reconfigured, the host does not recognize it anymore, and the host needs to be restarted because it can only can numerate the PCIe devices at boot up time.

For the application region, having AXI-Stream I/O interfaces was not very convenient when it wants to access the PCIe interface instead of UDP. Our initial idea was to have an adapter between the PCIe module and the application regime such as the AXI memory mapped to stream mapper core provided by Xilinx. However, this approach turned out to be impractical and clumsy. We decided to have two separate application modules with different interfaces for testing purposes.

6 Development Methodology

Since most of our modules were developed in HLS, we followed the standard Xilinx HLS development process. The processed was as follows:

1. Write the C code in HLS
2. Write the C HLS test bench for testing and verification
3. Optimize the C code for maximum hardware performance and efficiency
4. Optimize the generated hardware using HLS directives
5. Run C/RTL co-simulation to test hardware
6. Export the IP integrate the module in the system using Vivado system integrator

The optimization process is an iterative one so going through step 3 and 4 multiple times was necessary to make the design as efficient as possible.

6.1 Design Environment

For the FPGA platform, we used the 8k5 board provided by Alpha Data [5]. Vivado system integrator and Vivado HLS were our primary development tools. We used git on Bitbucket as source control tool.

6.2 Partitioning

The shell is a large system. Partitioning the project was vital to both dividing the project between the group members and make it easier to test and debug each sub-block individually. Figure 8 illustrates the partitioning of the projects. The project is divided into three main parts (from left to right): networking, memory interfacing, and application regime.

The networking consists of a UDP, TCP and Ethernet module along with a monitoring module to measure the network throughput. The second partition deals with the memory interfacing. It has a MIG to interface with the DDR4 memory outside the chip, bandwidth measuring module and a PCIe driver to send data back and forth between the host and the application. The third part is the reconfigurable application regime which is the min-sum decoder in our case.

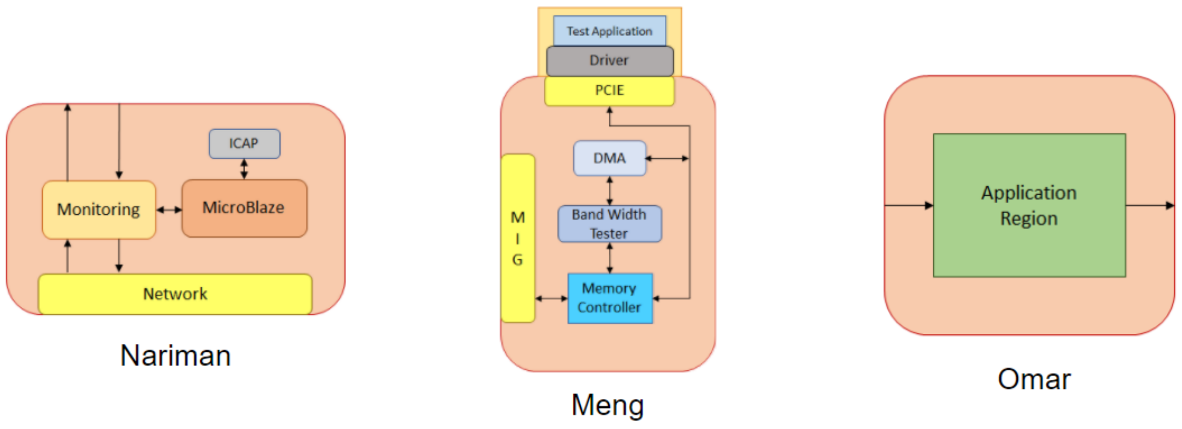

Figure 8: Main blocks of the shell. From left to right, memory networking, memory interfacing and application

6.3 Simulation, Verification and Testing

This shell implemented all the interfaces to communicate with a host outside the FPGA. To test different interconnections of the shell we have provided different tests (see Figure 9). For Ethernet testing (Layer 2), ping the FPGA is a practical way of testing. it will sends address resolution protocol (ARP) packets and get the response. To test higher layer of network, one application was needed in the FPGA, and one application was needed on the host to send and receive the data. In this regard, for the hardware side, we used Xilinx HLS cores. One UDP loopback application, and one TCP echo application was provided in Xilinx project. In host side, we developed a very simple application using python to send data to the FPGA, and get the result back.

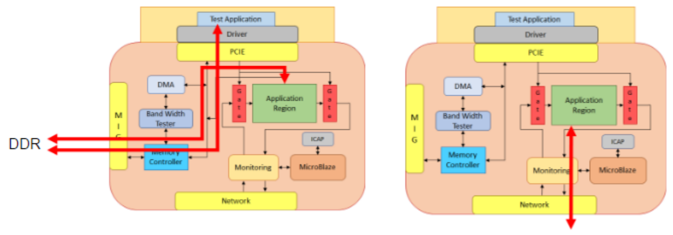

Figure 9: Left: test PCIe and DDR4 in the system. Right: network test and monitoring

In this shell hardware simulation was not straight forward because faking real network packets or inputs to PCIe and DDR memory is tricky. Xilinx has provided some test benches for these applications, but we used Integrated Logic Analyzer (ILA) and Virtual Input Output (VIO) to test the system. Using these debug tools, we could check signal values in the specific condition of the system.

To test the min-sum decoder module, we used the HLS testbench and compared the decoder's output with golden values generated from a similar min-sum decoder implementation in Matlab. The Matlab values were written to a file and the HLS testbench reads the golden values from the file and compares them to the HLS min-sum decoder's output. We also ran the C/RTL Cosimulation in HLS when we wanted to integrate the core into the system to check the sanity of the HLS hardware output.

7 Contributions

The partitioning tasks of the project is mentioned in Section 6.2. The contribution of each group member is as follows.

7.1 Nariman

- Implemented FPGA network backbone including Ethernet, UDP, TCP.
- Developed python scripts to test Network Layers.
- Implemented Memory Interface Using MIG for 8k5 Board DDR4 memory.
- Implemented PCIe DMA for used Platform.
- Design a Monitoring for the Shell.
- Integrated entire system.
- Developed a script to test the entire system.

7.2 Omar

- Implemented the min-sum decoder in HLS
- Optimized the implementation using HLS directives to reduce the total number of clock cycles
- Revised the decoder's interface and changed the module's design to handle those revisions

7.3 Meng

8 Design Characteristics

8.1 Resource Utilization

The resource utilization is available in Figure 10. This is the resource utilization after adding the application, with all the layers of network, two MIG for both DDR4 memory banks, PCIe interface, and Monitoring system. We observed that the TCP, utilize the big portion of recourses in comparison with other parts of the project, and without PCIe, it is less than 10% of FPGA.

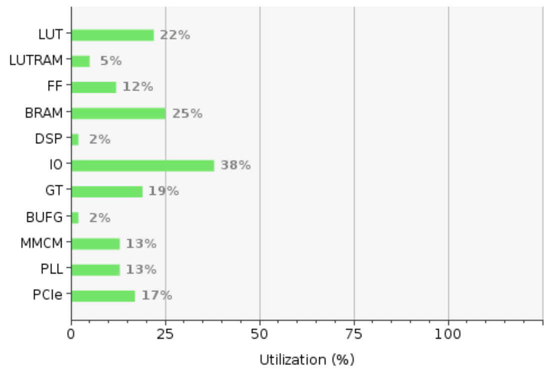

Figure 10: PAR report generated by Vivado

8.2 Where the Time Went

In design process of the shell, most of time went towards debugging TCP. The code provided by Xilinx was in Verilog, and that was hard to understand all the details. So, checking all the signals in that big system took a lot of time. This system has most of the available board interfaces in it. The other time consuming part was integrating different parts of the project into one big shell, and adding application to it. It is good to mentioned that each implementation of the system takes more than one hour to be completed.

At first, the min-sum decoder was written in an attempt to have the decoder up-and-running as soon as possible. There was some time spent in designing the circular shift memory modules necessary for storing the likelihood messages. Then, the design was refined from a C coding and hardware perspective to deliver the most efficient core in terms of clock cycles. Using HLS directive and other C coding trickery, we were able to reduce the number of clock cycles from $35,000$ to $10,000$. Note that the decoder uses floating point arithmetic which is not very efficient in terms of hardware. The min-sum decoder could be more efficient in floating point but we couldn't achieve better results due to time constraints.

9 Problems

The integrated TCP module in our shell has quite a few problem. The first problem was reseting the shell and application. In that big system, there are many clock regions and they should have proper reset. For example, there are two MIGs in this system, which need to be calibrated before the system starts to work. So we needed to first wait for the calibration to be be done, then restart the network and PCIe, and then restart the whole system and applications.

The other problem was about the Vivado versions. There is a bug in data flow model of HLS after vivado HLS 2015.1. So it caused quite a few problems. When we tried to wrap the min-sum decoder application using UDP interface. we could not use the implemented HLS core using vivado HLS 2015.1 in vivado 2016.3 because in opt design process, it complained about the floating point modules which were used in the HLS min-sum decoder core.

10 Experience with HLS

HLS is a versatile, user-friendly tool which allows even software developers to get on board hardware design on FPGAs. The directives for hardware optimization provided an even better grasp of which modules should be implemented in hardware and which kind of memory should be used. Fortunately, in some cases, HLS is smart enough to figure out what is the best hardware implementation for the code and applied directives automatically. The HLS design process was obviously more time efficient then designing hardware in HDL.

However, the downside of this level of abstraction is that sometimes, one might not be completely aware of the hardware generated by the tool. In some cases, such as implementing the network monitoring module, using HDL might have been more useful since it would provide a more granular hardware design.

An interesting workaround we needed to do in HLS involved converting `ap_uint<32>` to `float` bitwise without actually casting the variable. The standard C method is to use `union` where the memory location can be accessed as both types. However, `union` is still not supported in HLS. We needed to do a workaround which involved using a `void*` pointer to trick the compiler and circumvent the C casting. This workaround was necessary when receiving a UDP packet and feeding the data to the min-sum decoder application.

11 Conclusion

The demands for higher computation power is increasing rapidly. However, power limitation prevents the system to scale larger. In this regard, special purposes component could be useful because they are high performance and low power. One of these component is FPGA, which could be used in datacenter applications. However because of the hardship of designing for FPGA, and interfacing with other common resources in datacenters, they are not widely used. In this project by implementing a shell we addressed interfacing problem of these components, So the user could put his or her function into shell, and does not need to worry about the communications with the outside world. Using HLS could help a lot to implement a high-level and big design in a very short amount of time, for example, in this project, we used HLS to implement an LDPC decoder using the min-sum algorithm, and put it inside the shell as a proof of concept.

References

[1] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE Press, June 2014, pp. 13–24. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-reconfigurable-fabric-for-accelerating-large-scale-datacenter-services/

[2] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer - Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[3] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar, "High-level language tools for reconfigurable computing," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 390–408, March 2015.

[4] Xilinx. Tcp/ip project. [Online]. Available: https://github.com/Xilinx/

[5] A. Data. Alpha data 8k5 fpga board. [Online]. Available: http://www.alpha-data.com/dcp/products.php?product=adm-pcie-8k5

[6] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, January 1962.

[7] D. J. C. MacKay and R. M. Neal, "Near shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 33, no. 6, pp. 457–458, Mar 1997.

[8] F. Zarkeshvari and A. H. Banihashemi, "On implementation of min-sum algorithm for decoding low-density parity-check (ldpc) codes," in *Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE*, vol. 2, Nov 2002, pp. 1349–1353 vol.2.

[9] Creonic. Ldpc decoder applications. [Online]. Available: https://www.ldpc-decoder.com/