UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
ENGENHARIA DE COMPUTAÇÃO

ÁLAN FERREIRA DIAS

High Level Synthesis of a Min-Sum C LDPC Decoder

Final Report presented in partial fulfillment of the requirements for the degree of Computer Engineer

Prof. Dr. Axel Braun
Advisor

Prof. Dr. Renato Perez Ribas
Coadvisor

Porto Alegre, July 2014

“No, I’m NOT tryna be ya man, pimp bones in my body.

Rock them body-hotty, rock them, like ladi-dadi. - Nick Cannon”

ACKNOWLEDGEMENTS

I thank my tutor Dr. Axel Braun for conducting my work at the University of Tübingen. I would also like to thank Prof.Dr. Renato Perez Ribas for his help and suggestions which certainly added value to this work.

To Luiza Martins Couto I thank for the help and for the great times shared with me. I thank Rosangela Ferreira, my mother, and Maura Nogues Diaz, my grandmother, for all the support during my whole course and life.

I thank my friends Eduardo de Medeiros Gass - my brother from different parents - Rodrigo Hertzog, Mateus Dalepiane, Priscilla Rocha, Laurence Escobar, Lucas Schons, Cássio Mello, Luís Henrique Reinicke, Luis Henrique Grassi and everyone who was with me, for the friendship, for the help during the classes and for the all legendary stories together.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

AWGN	*Additive White Gaussian Noise*
BFA	*Bit-Flip Algorithm*
BP	*Belief Propagation*
DS	*Delayed Stochastic*
DUT	*Design Under Test*
DVB-S2	*Digital Video Broadcasting - Satellite - Second Generation*
FEC	*Forward Error Correction*
FPGA	*Field Programmable Gate Array*
HLS	*High-Level Synthesis*
LDPC	*Low-Density Parity-Check*
LS	*Logic Synthesis*
LUTs	*Lookup Tables*
MMGC	*Markov-Modulated Gaussian Channel*
MP	*Message-Passing*
PCM	*Parity Check Matrix*
PNR	*Place and Route*
RTL	*Register Transfer Level*
SAM	*System Architectural Model*
SP	*Sum-Product*

SPA	*Sum-Product Algorithm*
SoC	*System on Chip*
TB	*Test Bench*

LIST OF FIGURES

ABSTRACT

This work proposes to utilize the high-level synthesis technique to implement a LDPC (Low-Density Parity-check Code) decoder in hardware, starting with its software version.

Location and correction codes are essential in the computing world. The LDPC codes created in 1960 were rediscovered in the years 1990's and its utilization has become each day more frequently used in high-end technologies, due to its great performance.

However, implementing the decoder for such codes proved to be a great challenge for the hardware development community.

This work will present the steps utilized to synthesize hardware working the closest as possible from the behavioral model of system. The final results are created so they work in a Xilinx Virtex 6 FPGA.

Keywords: High Level Synthesis, LDPC codes, System Design, block codes.

Implementação em Hardware de Decodificador de códigos LDPC com Síntese de Alto Nível

RESUMO

Esse trabalho propõe utilizar o método da síntese em alto nível para implementar um decodificador de códigos LDPC em hardware, partindo de sua implementação em software. O decodificador sintetizado é voltado para aplicação em códigos de barra 2D.

Códigos de detecção e correção de erros são essênciais para o mundo da computação. Os códigos LDPC criados em 1960 foram redescobertos nos anos 1990 e sua utilização torna-se cada vez mais frequentemente utilizada em aplicações de ponta, devido ao seu alto desempenho.

Contudo, a implementação de decodificadores mostrou-se um desafio para a comunidade de desenvolvimento de hardware.

Serão apresentados os passos utilizados para sintetizar hardware trabalhando o mais próximo possível do modelo comportamental do sistema. Os resultados finais são gerados para operar em FGPA(Xilinx Virtex 6).

Palavras-chave: códigos LDPC, Projeto de Sistemas, Síntese de Alto Nível, HLS.

1 INTRODUCTION

Location and correction codes are of vital importance in the computing world. It is known that when we transmit data through any communication channel (telephone line or optical fiber, for example) it is susceptible to noise which can bring errors to the transmission. Every error correction technique is based on the concept of transmitting some kind of redundancy with the code, so it is possible to recover the original data in case that it is corrupted.

In 1948 Claude E. Shannon showed [1] that any communication channel can be characterized by two simple parameters: bandwidth and noise. Bandwidth is the difference between the upper and lower frequencies in a continuous set of frequencies, and may sometimes refer to passband bandwidth, depending on context. Passband bandwidth is the difference between the upper and lower cutoff frequencies.

The cutoff frequencies are defined as the frequencies in which the signal power drops down to half (3dB) and they are a feature of the transmission environment. Noise is any modification in the signal spectrum.

Given a channel with particular bandwidth and noise characteristics, Shannon showed how to calculate the maximum rate at which data can be sent over it without errors. He called that rate “the channel capacity”, but today, it is just as often called the Shannon limit.

However, during many years, the transmission rate achieved showed to be well below to the theoretical limit foreseen by Shannon, so a lot of research was done in order to find ways to transmit information in a rate closer to this limit. A group of engineers demonstrates that newly devised error-correcting codes could boost a modem transmission rate by 25 percent, but it was still far below to Shannon’s limit, it was necessary to develop codes that could increase the performance.

In 1963 Robert G. Gallager develop in his PhD Thesis [2] the theoretical base for Low-density parity-check (LDPC) codes. Due to its computational cost, too high for the time when the first commercial modem was just being launched, their incredible potential remained hidden and the LDPC codes were forgotten for almost 35 years. In the meantime the field of forward error correction was dominated by highly structured algebraic block and convolutional codes as Reed Solomon codes.

In 1993, Claude Berrou and Glavieux introduced the Turbo codes [3], the first to get closer to Shannon's limit. These codes became the standard in the late 1990s, showing great performance. As they were created basically with the trial-and-error method, researchers struggled through the 1990s to understand just why turbo codes worked as well as they did.

However, shortly after that, more research in this field led to the rediscovering of the LDPC codes in 1995 David J. C. MacKay and Radford M. Neal [4] in parallel with Niclas Wiberd [5]. Today the LDPC codes use is increasing in high-end technologies due to their performance that can usually easily overcome the Turbo Codes.

In addition to the strong theoretical interest in LDPC codes, such codes have already been adopted in satellite-based digital video broadcasting in 2003 beating six turbo codes to become the error correcting code in the new DVB-S2 standard, and long-haul optical communication standards LDPC is also used for 10GBase-T Ethernet, which sends data at 10 gigabits per second over twisted-pair cables. As of 2009, LDPC codes are also part of the Wi-Fi 802.11 standard as an optional part of 802.11n and 802.11ac.

LDPC codes are a class of linear block-codes, which means that any linear combination of codewords is also a codeword, and belong to the field of forward error correction (FEC) also referred to as channel-coding. FEC provides the receiver with the ability to correct errors without a reverse channel to request the retransmission of data. They are capacity-approaching codes, as practical constructions exist that allow the noise threshold to be set really close (in some cases arbitrarily close) to the Shannon theoretical maximum.

The noise threshold defines an upper bound for the channel noise, so below that the probability of losing information can be made as small as wanted. It has been shown that the performance of appropriately designed LDPC codes asymptotically approaches to Shannon limit as we extend the code length [6]. Using iterative belief propagation techniques, a message passing algorithm for performing inference on graphical models, LDPC codes can be decoded in a computing time which is linear on their block length.

1.1 LDPC Decoder

There are several decoding algorithms for LDPC codes that were discovered independently over time and ended up receiving different names. Robert Gallager had already provided a decoding algorithm when he first introduced LDPC codes, which is nowadays called belief propagation (BP), sum-product (SP) or message-passing (MP) algorithm. Wolfgang Proß has used LDPC codes in his PhD Thesis [7] to construct two dimensional(2D) barcodes targeting its use in industrial environments.

Figure 1.1 and figure 1.2 shows the two most famous barcodes. LDPC codes with short block length are addressed since the number of bits that can be stored inside of a 2D barcode is limited.

The feature of LDPC codes to perform near the Shannon limit of a channel is known for large block lengths only. So he introduced a new method for decoding short LDPC codes and proved that his method gives a superior decoding performance for additive white Gaussian noise (AWGN) channels and for the Markov-Modulated Gaussian Channel (MMGC) compared to well-tried optimization methods known from literature.

Figure 1.1: QR code.

Figure 1.2: DMC code.

1.2 Motivation

In the reading algorithm for 2D LDPC-based barcodes, the decoding of these codes is critically important. When the code is read, several factors (as variations in the reading angle or even dirty on the code, for example), increase drastically the probability of occurring errors. Furthermore, specially in the industry, this process needs to be done as fast as possible.

The work here presented was develop at the Eberhard Karls University of Tübingen inside a project that aims the decoding of 2D barcodes, as a way to accelerate the decoder initially developed in C++ by Wolfgang Proß in his PhD. For that, the decoding algorithm will be implemented in hardware.

1.3 Goals

To achieve that, the method chosen to generate hardware was high-level synthesis (HLS) in SystemC, a standard for hardware development that is growing every year and getting more and more acceptance in the industry.

In this work scope it will be done the less modification as possible in the original code trying to shrink the errors which are created during the process of hardware design, lower the development time, as well as studying the possibility of synthesize hardware starting with a software description.

Structure of the Technical Report.

This work is structured as follow, Section 2 gives a theoretical review of LDPC codes and the hardware synthesis through HLS. Section 3 discusses related works. Section 4 show the techniques used to synthesize the LDPC decoder of Wolfgang Proß in hardware. Section 5 presents the results gotten and finally section 6 gives the conclusions and indicates possibilities of future work.

2 THEORETICAL BACKGROUND

This section gives the reader a theoretical background on LDPC codes and Systems Design. It starts defining what are LDPC codes on section 2.1, then section 2.2 shows the data structure used for the decoding algorithm, the Tanner Graph. Section 2.3 gives the reader a basic idea how the decoding algorithm works. Section 2.4 gives a theoretical background on Systems Design.

2.1 Low-Density Parity-Check codes

LDPC codes are linear block codes, which means that every linear combination of codewords is itself a codeword. The notation and the definitions used here follows the approach in [7].

In the case of binary messages, the information is divided in blocks of bits with length k, what can represent until 2^k different messages $u = (u_0, u_1, \ldots, u_{n-1})$. For encoding m redundancy bits are added, so the message u becomes a new message x represented by n bits (with $n = k + m$). In linear block codes this means that the messages are:

$$x = x_0, x_1, \ldots, x_{n-1} = u_0 g_0 + u_1 g_1 + \ldots + u_{n-1} g_{n-1} \tag{2.1}$$

This can also be written in the matricial form as:

$$x = uG \tag{2.2}$$

G is a matrix of dimensions (k, n) and it is called the generator matrix, because the linear combinations of its lines with the bits in the message u as coefficients generate the new codeword x.

There is also a (m, n) matrix H that fulfils:

$$GH^T = 0 \tag{2.3}$$

Where 0 is an all-zero matrix with (k, m) dimensions. As x is a set of linear combinations of G:

$$xH^T = 0 \tag{2.4}$$

The matrix H of dimensions (m, n) is known as Parity-check matrix and the scalar product (also called dot product) with a codeword vector x must be zero. In the context of error correction, if this scalar product result is not zero then the codeword is wrong. In other words, each row in H does a parity-check on the codeword x received.

Thus linear code blocks are defined by their generator matrix G and their parity-check matrix H (PCM). The encoding part of LDPC codes is done using the Generator matrix and the decoding part uses the Parity-check matrix. LDPC codes are linear block-codes with low density of non-zero elements in their PCM.

There are two number that can be used to characterize this matrix: W_r is the number of 1's in each row and W_c is the number of 1's in each column. For a (m, n) matrix to be considered low-density it must fulfil two conditions: $W_c << m$ (that is, the number of 1's in a column has to be much smaller than the number of lines) and e $W_r << n$ (the number of 1's in a row has to be much smaller than the number of columns). From this definition comes the "low-density" part on the LDPC codes name as the PC part comes from the Parity-check matrix PCM.

2.2 Tanner Graph

The algorithm synthesized in this work utilizes the Tanner Graph data structure. The Tanner Graph was introduced in 1981 by Michael Tanner [8] as a graphical way to represent LDPC codes. It is a bipartite graph, which means that its vertices can be divided in two disjunct sets (an edge connects only nodes in distinct sets).

In the case of linear block codes, these sets are called Check-nodes (also known as Subcode-nodes or Parity-nodes) and Symbol-nodes (also known as Digit-nodes or Variable-nodes). The Check-nodes represent the rows of the Parity-check matrix and the Symbol-nodes represent the columns.

An edge connects a Check-node to a Symbol-node if the intersection between the column and the row at the PCM is non-zero. Figure 2.1 shows a Parity Check Matrix and figure 2.2 shows its corresponding Tanner Graph.

$$
\mathbf{H} = \begin{array}{ccccccccc|c}
\mathbf{S1} & \mathbf{S2} & \mathbf{S3} & \mathbf{S4} & \mathbf{S5} & \mathbf{S6} & \mathbf{S7} & \mathbf{S8} & \mathbf{S9} & \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{C1} \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \mathbf{C2} \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \mathbf{C3} \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \mathbf{C4} \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & \mathbf{C5} \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & \mathbf{C6}
\end{array}
$$

Figure 2.1: Parity Check Matrix (PCM) example.

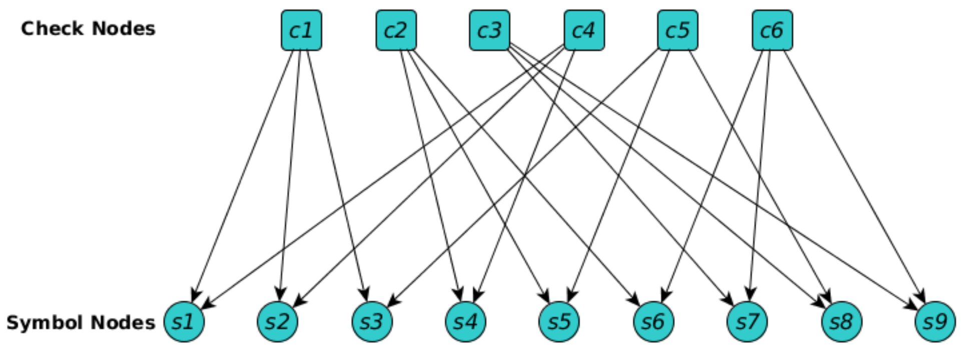

Figure 2.2: Tanner Graph.

Figure 2.1 shows a regular LDPC code with $W_c = 2$ and $W_r = 3$. Looking at the Tanner Graph it is also possible to observe that it is a regular code as the number of edges connecting every Check-node to its Symbol-nodes is always constant. When the number of 1's in a row or column is not constant in a LDPC code it is called irregular.

2.3 Decoding

The decoding algorithms are generally called message-passing algorithms as they can be explained as a set of messages being exchanged between Symbol-nodes and Check-nodes in a Tanner graph. A node on the graph uses only the information given by the nodes connected to it. A basic decoding algorithm is done as follows:

Step 1: Every Symbol-node S_i sends a message to their Check-nodes (always two in the figure 2.2) containing the bit that they believe to be the right one.

Step 2: The Check-nodes C_i calculate a response for each Symbol-node connected. The message at the answer contains the bit that C_i believes to be the correct for this Symbol-node S_i, assuming that the other Symbol-nodes connected to C_i are correct. In other words, using the example in figure 2.1, a Check-node Ci looks the received message and considering its parity-check equation, uses two Symbol-nodes and calculates the bit that the third Symbol-node should be to fulfil the parity equation.

Step 3: The Symbol-nodes receive the messages from the Check-nodes and uses this additional information to decide whether the bit that they received in first place was right. A simple way to do so is with voting. In the example, every Symbol-node has three sources of information, the original bit received and two suggestions from twi Check-nodes. The result is resend to the respective Check-nodes for the next iteration.

Step 4: Each Check-node is checked verifying if the parity equations are fulfilled. If yes, it means that the right word has been found. If not, in case there are still iterations to be performed it goes back to step 2, otherwise the decodification ends. Figure 2.3 ilustrates the algorithm.

This algorithm is useful for us to understand the basic ideas behind the decoding algorithms for LDPC codes. The message-passing algorithms are also called iterative decoding algorithms as the messages are passed between Check-nodes and Symbol-nodes iteratively until the correct result is found or the max number of iterations reached.

Different message-passing algorithms receive their names based on the type of message passed within them. Bit-flipping algorithms use binary messages, in the case of the algorithm developed by Wolfgang Proß [7] the technique used is belief propagation, which receive this name because the information transferred is the probability of a bit to be right. Usually it is convenient to express this probability in the Log domain.

In that way, the belief propagation algorithms are also known as sum-product decoding since using the Log domain allows the calculation be performed as sums and products operations. The algorithm synthesized in this work is called Min-Sum C decoder and uses a variation of the belief propagation technique, it uses more advanced math which has an explanation that is beyond the scope of this work.

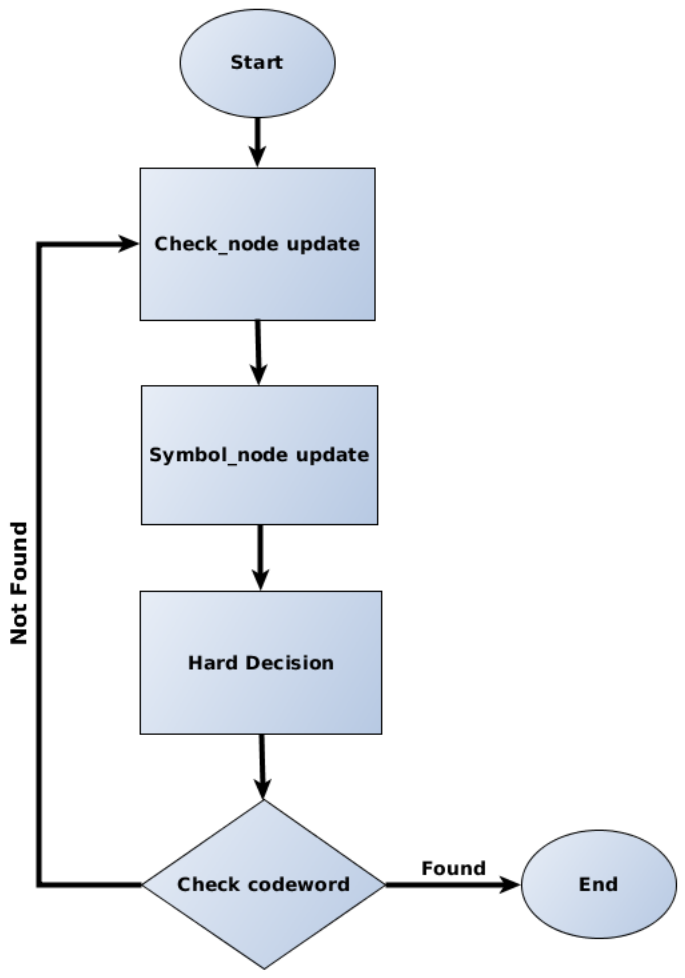

Figure 2.3: Decoding Flowchart.

2.4 Systems Design Process

Systems design is a very complex task. Usually this task implies the development of a basic system model in some language as C/C++, Java or similar. This model is called System Architectural Model(SAM). SAM is used as a communication channel between the algorithm, hardware and software groups in charge of the system development.

It can be used as a way to refine the chosen algorithm or as a basis for deriving a specification for hardware and software subsystems. After this step, typically each group uses its own specific language to refine their part of the system and the SAM is abandoned.

In the system design sight, this method can be basically represented in figure 2.4.

Splitting the development teams up and changing the languages introduces a great number of errors, especially in the transcription of the original algorithm written in a higher level language to hardware. Furthermore, this disconnection between the used

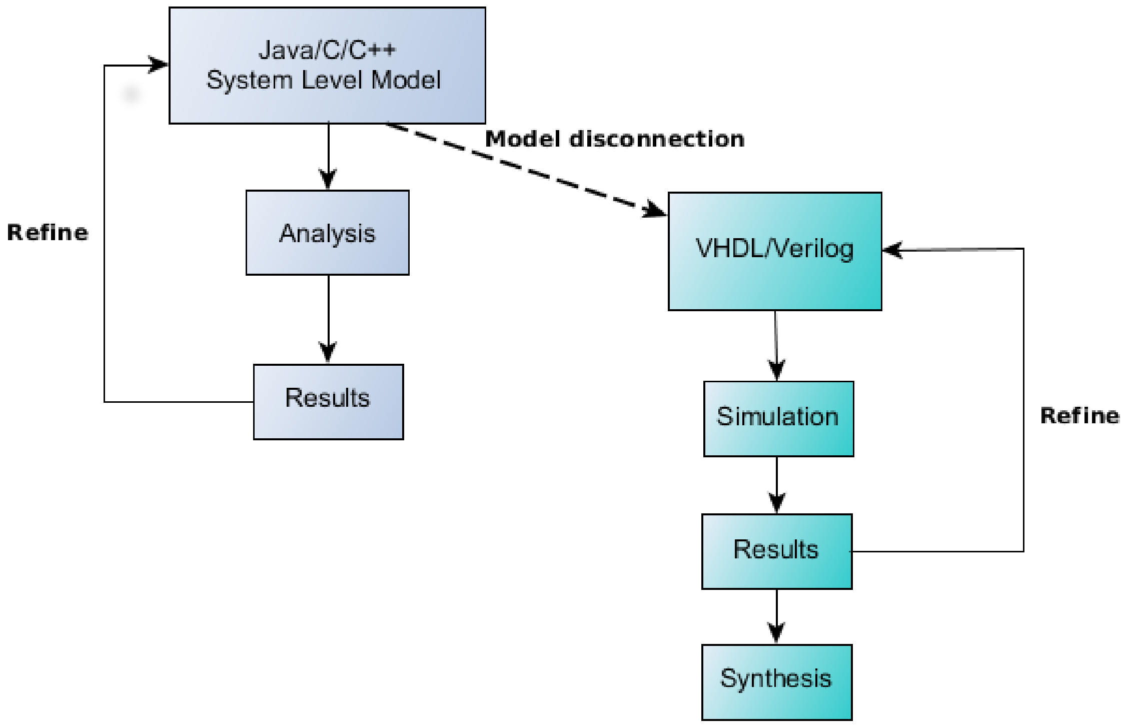

Figure 2.4: Classic System Design model.

models creates different testbenches for each version, that is, different ways to verify the system.

These are some of the problems associated with this classical system design method, which is still vastly utilized. Moreover, more complex and bigger systems are being developed everyday, which means more code lines and more implementation errors.

The complexity at the hardware development process implies its hierarchical separation in a set of design levels. A fundamental way to achieve that, is using information abstractions. An abstraction level in the context of computational systems design is a set of descriptions with the same detailment grade. For designing a System on Chip(SoC), a great number of design descriptions with different abstraction levels are used.

For instance, logical gates representing an adder have much less details than the physical layout of it, but both are relevant at some point in the design flow. Some abstraction level used are:

Architectural: Used for non-functional decisions, describe the system with interconnections between smaller subsystems.

Behavioral: Describes the system functionality without technological details.

Register Transfer Level (RTL): Implements the system as a set of registers and combinational logic, focus on describing the flow of signals between registers and is currently the most used model for hardware design.

Gate: The system is described in terms of logic gates and the interconnections between these gates.

Switch level (Transistors): The design is implemented using interconnected switches/transistors.

Physical Level (Layout): It is the lowest abstraction level. The circuit representation of the components of the design are converted into geometric representations of shapes which will implement the design in its physical form.

Figure 2.5 shows how the abstraction models work together:

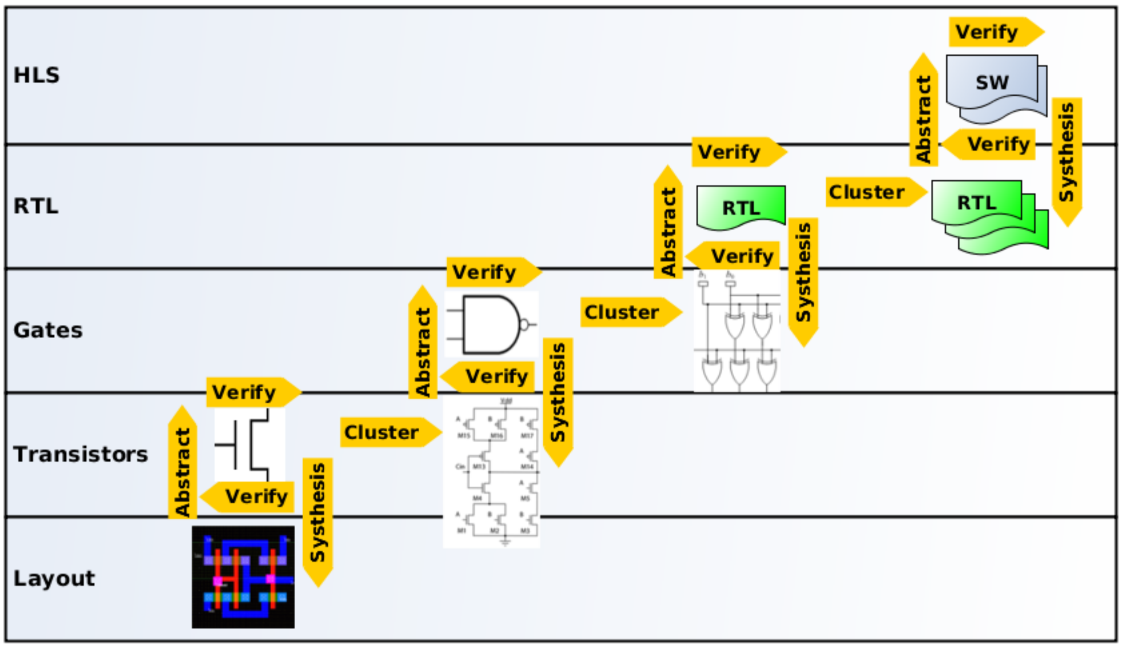

Figure 2.5: Abstraction Levels relation.

The number of abstraction levels needed during the design of complex systems is increasing along with the increasing complexity of these systems. Beyond that, the need to rise the productivity and to shrink the design time has made new abstraction levels higher than RTL emerge.

Figure 2.6 shows three examples of design generations for a SoC (System on chip): yesterday, today and tomorrow. It shows the number of code lines estimated for design this system at four abstraction levels. Tomorrow's design are being developed today, it is visible the exponential growth of code lines.

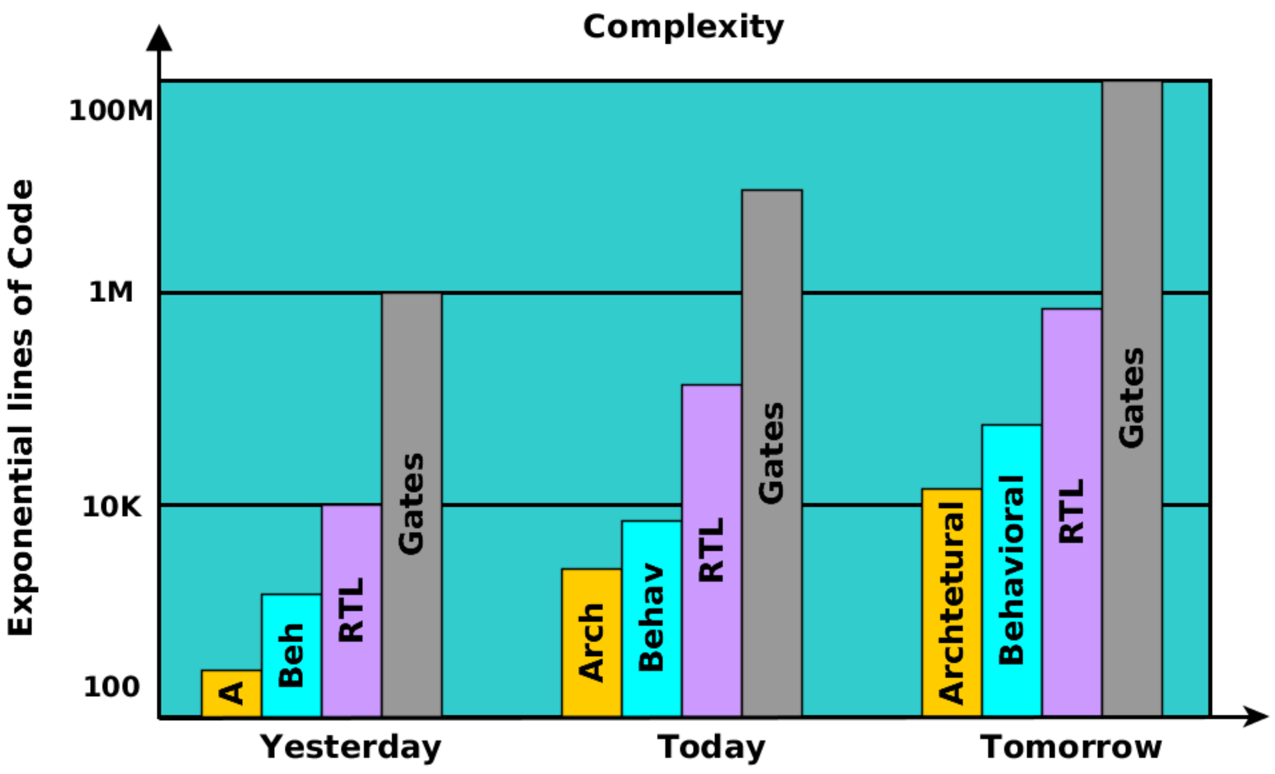

Figure 2.6: Code Complexity at Four Abstraction Levels.
Figure adapted from [9].

Even today's systems are only possible because there are a lot of tools developed to generate gates and layout automatically using the RTL design. Tomorrow's systems will easily have more than 1 million code lines of RTL code if we design them with the same techniques as before. These problems made it necessary for the design community to develop higher abstraction levels, new methods that will lower the errors manually introduced in the transcription out of the SAM model to the final hardware, lower the number of code lines needed and also shrink the design flow time.

For that, High-Level-Synthesis (HLS) was developed. HLS allows us to design hardware out of its behavioral implementation. Thus, when the project design is done at higher abstraction level, much less details are needed and less code lines used. Using less code lines, less errors are introduced and the repairs become easier. When a designer works at the behavioral level, he does not need to worry about issues as clock frequency or target technology. It makes possible for the designer to focus at the desired functionality.

Once the high level model is ready, HLS automatizes the process of RTL implementation, what eliminates potential manually introduced errors. The generated code is then passed to one of the long-term used tools to synthesis that uses the RTL model as input. A language that already proved to fulfil the requirements for this task and is becoming each day more important for for HLS is SystemC.

Strictly speaking, SystemC is not a language, but a library of C++ classes which implements the functionalities needed for HLS modeling, however it is constantly quoted as a language by its own. Many tools are available today that allows hardware synthesis using SystemC. Trying to take advantage on these HLS characteristics, the LDPC decoder that this work describes is going to be implemented in SystemC utilizing the Cadence's tool Cynthesizer version 4.1.

3 RELATED WORK

High Level Synthesis has been an intensive area of research for a long time. In [10] published more than 20 years ago, Raul Camposano explains how the idea of HLS is even older, having started in the years 1960. However, it was seen only as an academic activity as the real synthesis problems at the time were located in others abstraction layers, as the physical layer.

The author also described in [10] the steps made in HLS for it to work. He showed how the knowledge basis for utilizing it were already in a state where many practical applications could be done, with losses in both area and performance. The maturation of HLS tools and their acceptance by the industry, however, took much more time and only now this kind of synthesis seems to be starting its consolidation on the industry.

Nowadays, according to the authors in [11], for the Field Programmable Gate Array (FPGA) design community HLS is transitioning from a field of research and investigation to the development method choose. Also in [11], the authors make an analysis of the various advantages and disadvantages of this method and, in the end, explain some of the problems that are not solved yet.

Nevertheless, many tools were developed so that HLS could become a reality. In [12], the authors made a comparison between several tools of this method that are available today, evaluating them under various aspects. As already said in [11], it is demonstrated that, while these tools had greatly evolved in the last years and its utilization by the industry is already possible - and a reality - a lot of work still need to be done.

David J. Pursley and Brett L. Cline [13] tried to introduce HLS as a way to measure the trade-offs of algorithm implementation in hardware or software. The tools utilized for this is the same used in this work, the Cadence Cynthesizer. The authors came to the conclusion that using HLS can be useful to decide whether an algorithm will be implemented in hardware or software. They notice that the method can even bring great performance gains with a small incrise in area. These results are encouraging because formerly using HLS would bring the certainly of performance loss with area increase.

Furthermore, in the error correcting field, great efforts are being made upon LDPC codes. Besides the different codes and different decoding algorithms proposed, a lot of implementations for LDPC decoders have been proposed, as the decoder implementation proved to be the bottleneck to implementing LDPC codes in hardware.

In this way, various LDPC decoder architectures can be find in the literature. The slowest one, is the bit serial approach [14], but it is also the cheapest in terms of the hardware resources needed. The fastest approach is the fully parallel one [15] [16] , where the whole Tanner graph is transferred to hardware. Although this approach can provide very high decoding throughput, it also requires a much larger amount of hardware resources.

Another approach used are the partially parallel decoders [17] [18] which uses just a few number of variable and check node processors in order to do some of the processing in parallel. These decoders achieve lower throughputs than fully parallel architectures, but consume much less hardware resources specially due to the easier wire routing.

Vikram Arkalgud Chandrasetty and Syed Mahfuzul Aziz [19], implemented in a FPGA a LDPC Decoder using a Reduced Complexity Message Passing algorithm for large LDPC codes. Their main idea was to join the efficiency of the Sum-Product algorithm with the simplicity of the Bit-Flip algorithm (BFA). The decoder was developed in Verilog. The proposed algorithm has reduced the average number of decoding iterations compared to BFA and the decoder requires less hardware resources compared to the Sum-Product algorithm (SPA).

Jin Sha, Minglun Gao, Zhongjin Zhang, Li Li and Zhongfeng Wang, developed in [20] a Quasi-Cyclic Low-Density Parity-Check decoder in a FPGA utilizing a the Verilog language. The decoding algorithm used was a Modified version of the Min-Sum algorithm to reduce the memory size needed for information storage.

Another approach was made by Mohammad M. Mansour and Naresh R. Shanbhag [21], they jointly designed the code and the decoder seeking to lower the computing complexity and the power consumption as well. To do so, they proposed to partition the PCM in smaller matrices what would simplify the interconnections between check nodes and symbol nodes.

Hao Zhong e Tong Zhang presented a partially parallel LDPC decoder [22], trying to lower the complexity of the routing wire in large LDPC codes. The decoder developed utilizes codes of length 4K and 8K. The main advantage is that it was possible to explore the trade-off between the hardware complexity and the decoding speed.

In [23] the authors presents a hardware architecture for a fully parallel stochastic low-density parity-check decoders. This decoder utilizes an irregular LDPC code (1056,528) and it is implemented in a FPGA. The results provided in this paper validate the potential of stochastic LDPC decoding as a practical and competitive fully parallel decoding approach.

In [24] the authors introduced a new stochastic algorithm called Delayed Stochastic (DS) decoding, to implement low-density-parity-check decoders. The algorithm proposed uses an alternative method to track probability values, which according to the authors results in reduction of hardware complexity and memory requirement of the stochastic decoders. Two decoders are implemented using the DS algorithm for medium (2048, 1723) and long (32768, 26624) LDPC codes, using a fully-parallel implementation of these decoders.

Alexios Balatsoukas-Stimming and Apostolos Dollas [25] implemented a fast fully parallel FPGA-based design of LDPC codes using a code length of 1000 and 1152. They have used specific FPGA optimizations for shrinking the Lookup Tables (LUTs) need. The final throughput is high, but the architecture lacks flexibility.

The best implementations achieved good performance but little flexibility, reaching high frequencies only with platform-specific optimizations and using large LDPC codes.

For contribute with these efforts, this work proposes to implement the decoder described in [7] for small LDPC codes, developed to be used with 2D barcodes. Using the HLS method will help to lower the development complexity and to make the project more flexible.

4 IMPLEMENTATION

4.1 Goals and Strategy

Before starting out with the project, it is first needed to consider what we are trying to accomplish. As previously said, the final goal of this work is to implement in hardware the LDPC decoder developed in C++ by Wolfgang Proß [7]. The High Level Synthesis method will be used so it is possible to keep the project the closer as possible to the decoder algorithm model.

For that, first we must build a high-level C++ model of the LDPC decoder that we wish to implement. This must include a test bench to ensure that the functionality incorporated in the model behaves as expected. This high-level C++ description of the design will be converted to RTL Verilog by the tool. So we can perform simulations of all generated models (RTL C++, RTL Verilog, and Verilog gates) against the original C++ test bench in order to verify that the transformations were done correctly.

4.2 Key files

The key files of the project are:

dut.cc + dut.h: Source code of the module. This is the core functionality that will be synthesized.

tb.cc + tb.h: The testbench for the design module.

project.tcl: A Cynthesizer project file that contains the definition of the project and all of the different explorations that will be performed. It is used to automatically generate a Makefile that will be included in the top-level Makefile .

Makefile: A Makefile which contains all of the rules required to control the various Cynthesizer runs for the project.

goldenfiles: A directory that contains a number of “golden” results files to validate that all versions of synthesized code are still functioning as expected.

A number of other support files are included in the project directory. These files will not be described in detail.

4.3 Project Structure

Basically, a system in SystemC is constituted of two main parts: the Design Under Test (DUT), with is the device desired to be implemented, and the Test Bench (TB), which is used to give the input data to the DUT and to receive the output that will be verified later.

The DUT can be divided in several smaller modules if desired. Figure 4.1 illustrates this configuration.

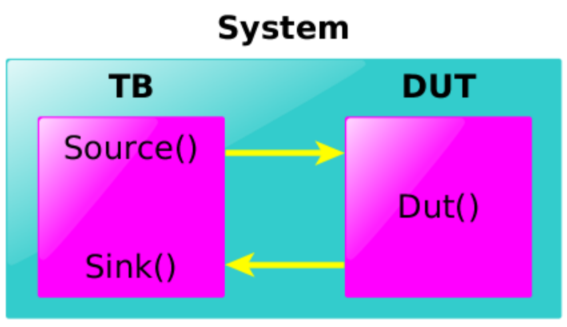

Figure 4.1: System in SystemC.

4.3.1 DUT Structure

The basic structure of the dut is as follows.

```
void dut::mainThread(){

 reset();

 while(true){

    dataIn();

    //DUT functionality here.
    decoder();
    dataOut();

  }
}
```

reset() - DUT reset code: initialize all output ports and all data members of the module.

dataIn() - input hardware interface: This function represents the input interface of the module. It is used to get data from the TB.

decoder() - Function where the actual code working on the input data is placed.

dataOut() - output hardware interface: This function represents the output interface of the module, used to send data to the TB.

The DUT functionality should be placed between the dataIn() and dataOut() functions which are placed inside an infinite loop.

4.3.2 TB Structure

The basic structure of the TB has only two functions that run in parallel.

source(): Code to get the input for the DUT and drive the DUT input ports with the data. Here is also the reset function for the TB;

sink(): Code to receive the outputs from the DUT, process the output data and compare it with the golden files. It is also used to end the simulation calling the function esc_stop().

4.4 Golden Files

The adapting process of the algorithm from software to hardware implies modifications in the original code. The code below shows the main decoder loop.

```
for (int i = 0; i < maxIter; i++) {
    //Check-nodes update
    CN_update();

    //Symbol-nodes update
    SN_update();

    //hard decision
    hd();

    // Check if valid CW found. If found, ends algorithm
    if (check_CW()) break;
}
```

Methods are needed to verify that the project developed is still reproducing the same results as the original algorithm. Typically, to check if the algorithm is working, files with the desired output are used to compare with the output generated by the project.

Usually, these data are obtained resolving the algorithm "by hand" or consulting a specialist who can give the desired output. In this work, it is necessary to obtain the same results as the software version. So, in order to get this output, the original software was modified so it saves its output in a text file.

Besides that, as the LDPC codes decoding algorithm utilizes a set of steps and message exchanging between symbol and check nodes at the Tanner Graph, it is important to know if the intermediate steps are also providing the right results.

Because of that, all the data in these steps were also saved in separated files, willing to ensure that the modified hardware version is providing the same data. This was done modifying the CN_update(), SN_update() and HD() functions so they salve their outputs to the golden files.

4.5 Division between Software and Hardware

After the verification data was obtained, the code modification to the SystemC version can be made. Deciding what is going to be implemented in the Test Bench (which will continue to be software) and what is going to be implemented in the DUT (and will be synthesized to hardware) is one of the first project decisions that has to be done and is nothing trivial.

As mentioned in section 3, there are two main ways to implement the LDPC decoder in hardware, partially parallel and fully parallel. In order to obtain a decoder that can be optimized to high throughput, it was chose to use the fully parallel implementation, instantiating the entire Tanner Graph and the calculation in hardware.

It is known that the LDPC codes are defined by its Parity Check Matrix (PCM). This matrix is described in a file and this file can be written in several ways, but the information contained in it must be correctly passed to the DUT. Willing to make the project more flexible, the file describing the PCM will be read in the TB, this way the reading becomes liable of modification if we have a new structure of PCM file. Besides that, reading the file in the TB makes this step simpler.

The TB also sends the input word to the DUT and gets back all the data generated during the decoding. In the end, it also compares these results with the golden files.

4.6 Communication between TB and DUT

The data read in the TB must be send to the DUT some way, this characterize another project decision, which is implementing the communication between TB and DUT. When developing projects in High Level Synthesis, the TB should be the same for all the abstraction layers involved (starting in behavioral down to physical), for that, one must guarantee that the communication do not depend on details of the lower abstraction levels (as clock frequency, or waiting cycles for example)

Therefore, a handshake algorithm between these two modules was used, as it makes the communication independent of these parameters. Figure 4.2 shows the signals connections TB and DUT. A communication channel called inp_sig and two control signals called inp_vld and inp_rdy where used to send the input data to the DUT. In the same way, another communication channel called outp_sig and two control signals called outp_vld and out_rdy where used to receive the output from the DUT.

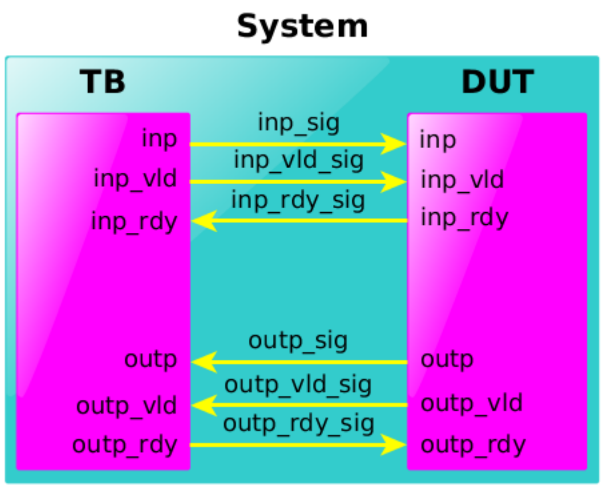

Figure 4.2: Communication signals.

The communication algorithm works identically for both ways (from TB to DUT and from DUT to TB). As an example, it will be shown the communication algorithm working from TB to DUT.

It works in three steps:

1. The TB put the data in inp and actives the inp_vld signal, indicating that there is a valid data to be read in the channel.

2. The TB waits a undefined amount of time until the DUT reads the data in the inp channel.

3. When the DUT reads the data in the inp channel, it sends the inp_rdy signal to the TB, indicating that the data was correctly read and it can moves to the next data.

Figure 4.3 illustrates this process.

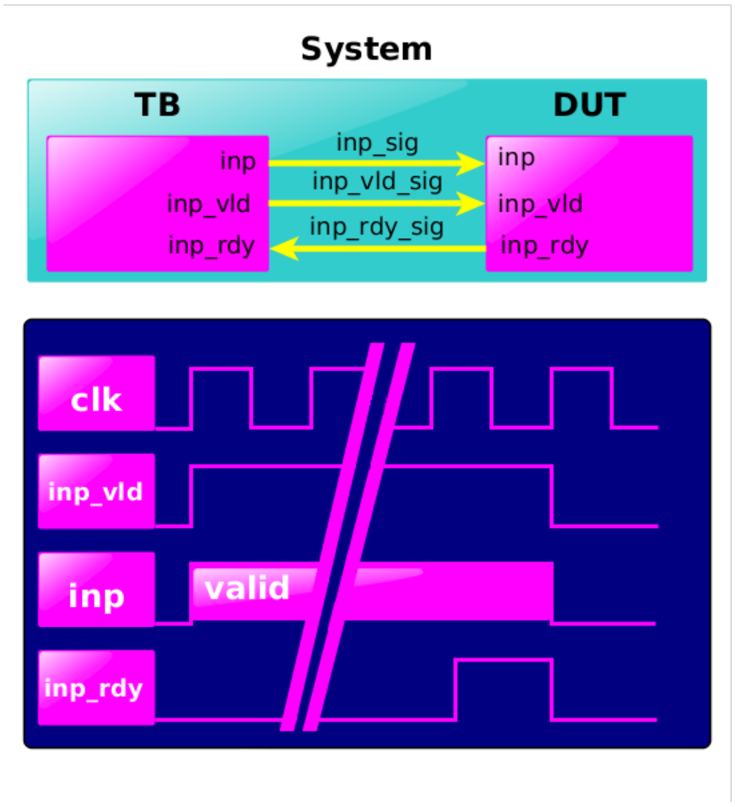

Figure 4.3: Handshake algorithm between TB and DUT.

4.7 Structures Used

After defined the communication, the input data can be send from the TB to the DUT utilizing the handshake described. These data are stored in a structure called TannerGraph in the same as it is done in the software version.

Structure description:

```
//Structure st_Node
//Used for each Symbol-node and
//Check-nodes contained in the st_PCM structure

typedef struct {
  int numRefs;       // Number of Neighboring Nodes
  int *Refs;         // References to Neighboring Nodes
  int *crossRefs;    // Ref-index of neighboring node
                     //where this node can be found.
  double *Messages; // Incoming messages from Neighboring nodes.
} st_Node;


//Structure st_PCM
//Used for the Tanner-Graph

typedef struct {
  int numSymbolNodes;    // number of symbol-nodes in the Tanner-Graph
  int numCheckNodes;     // number of check-nodes in the Tanner-Graph
  int numEdges;          // number of edges in the Tanner-Graph
  st_Node *SymbolNodes; // pointer to the symbol-node structures
  st_Node *CheckNodes;  // pointer to the check-node structures
}st_PCM;
```

The Node structure contains the number of references to its neighboring nodes (int numRefs), the references to these nodes (int *Refs), the reference index from the neighbor node where this node can be found (int *crossRefs, message to be send to neighboring node Refs[i] is stored in Neighboring_node.Messages [This_node.crossRefs[i]]) and the messages from the neighbor nodes for this node (double *Messages, Messages[i] is received from neighboring node Refs[i]).

The TannerGraph structure has a variable to store the number of Symbol-nodes in it (int numSymbolNodes), a variable for the number of Check-nodes (int numCheckNodes), a variable for the number of edges in the Graph (in numEdges) and two node lists, one for Symbol-nodes(st_Node *SymbolNodes) and another for Check-nodes (st_Node *CheckNodes).

4.8 Code Modifications

4.8.1 Modifications in the Structures

The structures described before contain dynamic elements list, that is, their size is variable and defined during run time. However, in the hardware version, there has to be a fixed size, as these lists will be transformed in memories during the synthesis process and it is necessary for the tools to know the memory size that will be allocated.

So, the structure came to be defined as follows:

```
//Nodes
typedef struct{
    int numRefs;
    int Refs[MAXREFS];
    int crossRefs[MAXREFS];
    double Messages[MAXREFS];
}st_node;

//TannerGraph
    int numSymbolNodes;
    int numCheckNodes;
    int numEdges;
    st_node SymbolNodes[NUMSYMBOLNODES];
    st_node CheckNodes[NUMCHECKSNODES];
//
```

The chosen size for these lists is MAXREFS = 252, NUMCHECKSNODES = 252, NUMSYMBOLNODES = 504, MAXEDGES = 4096, because these are the biggest values found in the PCM file that describe these codes. This will guarantee that the decoder works for codes with this size or smaller (with some more modifications that will be described in the following sections).

Besides that, as there is no need to define the TannerGraph structure, as it is the only one in the whole code and it is global to the DUT, it started to not be defined as a data type anymore. While this change seems to have no impact, only by defining the Tanner Graph as a data type was creating extra control logic in the hardware synthesis and creating an unnecessary delay.

4.8.2 Modifications in the Control Variables

To store data in the structures, reading loops are used, as it is done in the software version. However, again some modifications must be performed. When reading the PCM file in software, loops with variable size can be defined, as the PCM file has the number of Check-nodes and of Symbol nodes and these values can be used to control the reading loops size.

Some HLS tools can deal with variable loops and Cynthesizer is one of them. Nevertheless, usually it is not interesting to use this ability. During the optimization part, it is normally useful do loop-unrolling in some loops, what cannot be done when the loop size is variable. In the limit, the tool can be set to unroll all loops in the project, although it is usually prohibitive due to the great increase in the area need to implement the design.

For this reason, the loops that were written this way:

```
for(int CheckNodes i = 0; i < numChceckNodes; i++){
    TannerGraph.CheckNode_Refs[i] = inp.get();
}
```

Started to be written this way:

```
for(int CheckNodes i = 0; i < NUMCHECKSNODES; i++){
    if(i == numCheckNodes) break;
      TannerGraph.CheckNode_Refs[i] = inp.get();
    }
```

This way, it is guarantee that the loops can be unrolled if it is interesting for performance reasons and, at the same time, they will work for cases where the PCM describes different codes (with the condition that the number of check nodes and of symbol nodes is smaller than the defined max).

After the data is correctly store in the structures, the next step is to ensure that all the calculation part work correctly. When passing the algorithm to the DUT functionality, one of the first problems to deal with is the fact that some functions normally used in C are not available in SystemC, that is, they are not synthesizable. They can be used for testing the behavioral version, but they must be replaced when we want to work in lower abstraction levels. This also applies to some data types (as double),

Functions to print data normally used in software (as prinft) must be taken out or passed to the TB. The function abs(arg) that returns the absolute value of arg was also not synthesizable and a SystemC version had to be implemented. When done with the replacement of these functions, the behavioral model was tested and validated. The comparison between the output files and the golden files showed that the results were identical, but to be synthesizable, more modifications had to be done.

4.8.3 Modifications in the Data Types

Not all constructors used in C/C++ are synthesizable with HLS, for example: alloc/-malloc, exit, new, recursive functions, union, bit fields and virtual functions.

Floating point variables initially are not synthesizable, but they can be used as parameters in some cynthesizer commands. These variables and function parameters are also supported as long as their values are not used in behavioral code that will be synthesized (used in the TB, for example).

Although there are some paid libraries that make it possible to synthesize floating point variables, this usually is translated to a huge increase in the area needed. For these reasons all the double typed variables where replaced by SystemC native variables for fixed point sc_fixed $< n, m >$, where n in the total bits number and m in the number of bits in the integer part. In this project, the main variable with the double type was Messages[MAXREFS], used in each node so it knows the messages from its neighbors. Testing the algorithm several times against the golden files, the value $< 62, 6 >$ proved to be sufficient so that the algorithm works without errors.

A lot of area is wasted when more bits than the necessary are used to store variables. For instance: a variable of the int type used to control a for of 10 iterations could be stored with only 4 bits using sc_uint<4>. However, if int is used, although the size depends on the platform used, usually the value is 32 bits. Because of that, all variables utilized in the design were replaced by native SystemC types. This not only decreases the area needed to implement the project as is makes it faster.

4.9 Libraries

In order generate the RTL model it is necessary to utilize a parts library which characterize the components for a specific technology. In this library are all the components(multipliers, adders, logic gates etc) parameters, as delays and area.

These libraries can be automaticaly generated using Cynthesizer version 4.1.

In this project, a part library for FPGAs were generated. The FPGA chosen is the Virtex 6, part XC6VHX565T, package FF1923 with speed grade -1.

5 RESULTS

Finally, with all the adaptations done, several representations on different abstraction levels were created so they can be tested against the TB developed. The representations used were:

BEH: Fully behavioral model, keeps the code in C. It was heavily used to test the code functionality at a high level, and it is really useful as it has the fastest simulation time.

RTL_C: RTL model generated in C++, the simulation is slower.

RTL_V: RTL model generated in Verilog, the resulting code is used as input for the other synthesis tools.

LS: Logic Synthesis, it used the resulting code out of the RTL_V representation as input to generate the gate-level model. This is done with the Xilinx SynplifyPro tool.

PNR: Place and Route, last synthesis level of this work. Uses the result from the LS representation as input and it is also done with the Xilinx tools.

For each representation a simulation was created and then tested against the TB, which compares the results with the golden files. All data generated by the DUT was exactly the same as the software version, with no differences in the golden files, what characterizes the correct functionality of the project.

The Appendix D presents the project.tcl file used in this project. The Appendix A, presents the main part of the tool log file showing the components generated to implement the RTL model. It uses a total of 3735 LUTs, 2 dedicated multipliers, 1634 register bits and 15 internal RAM blocks.

For the logic synthesis, which generated the gate-level representation of the design, the input was mapped to 2135 look-up tables (LUTs), 1447 register bits, 524 Block RAMs, and some dedicated cells. More details can be found in the Appendix 2. The Place and Route got the gate-level components mapped to 805 Slices in the FPGA (where each slice contains four 6-input LUTs and eight flip-flops), utilizing 1977 LUTs and 1447 registers, 524 Block RAMs and other dedicated components, as the logic synthesis. More details are placed in the Appendix 3.

As no implementation of this specific algorithm was found in the literature, it was chosen to compare it with algorithms that use a similar word size (n, k) - where n in the total number of bits and k in the number os redundant bits. The design uses a (756,252) regular LDPC code, as it is the word size in the software version.

The design was compared to other works from [19] and it shows great results, using less area and components as it shows in table 5.1 (the values used were the ones obtained after the place and route).

It is also important to note that the design also addresses one of the main weakness of the other projects which is the lack of flexibility.

Resources	SPA[19]	SMPA[19]	BFA[19]	Min-Sum C (This work)
Device	Virtex 5(XC5VLX110T-3FF1136)			Virtex 6(XC6VHX565T-1FF1923)
LDPC code	(648,324) regular			(756,252) regular
Slices	15684	4046	1396	805
LUTs	58787	14239	3577	1977
Registers	12443	5963	2069	1447

Table 5.1: Comparison with other works

6 CONCLUSIONS AND FUTURE WORK

This work used a combination of techniques and principles acquired during the course of Computer Engineering as computers architecture, digital circuits and programming.

The project was successful, as all the goals were achieved and validated within the scheduled. The set of techniques here used can be applied to any kind of project where it starts with an algorithm in a high level language and it should be implemented in hardware using HLS.

Besides that, it can be concluded that the HLS has reached a maturation level which allows complex hardware implementations to be generated starting out with models much closer to the algorithmic model, what brings great benefits to the developers and the industry, as discussed before.

The work developed here has several options to be continued, starting with optimizations that the tool itself provide. Among them the pipelining and the loop-unrolling techniques can be highlighted. Another modification that which could show some good results is introducing an explicit memory to the project, although this would also create the necessity to control it manually.

It can be noted that although the HLS tools are currently very powerful and efficient in the systems design, the designer still needs to deal with architectural issues and take specific decisions for the hardware design.

REFERENCES

[1] Claude. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, July 1948.

[2] R. G. Gallager. *Low-Density Parity-Check Codes*. PhD thesis, Massachusets Institute of Technology, 1963.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. *IEEE Proceedings of the International Conference on Communications*, 2/3:1064–1070, May 1993.

[4] D. MacKay and R. Neal. Good codes based on very sparse matrices. *Cryptography and Coding*, pages 100–111, December 1995.

[5] N. Wiberg. *Codes and Decoding on General Graphs*. PhD thesis, Linköping University, 1996.

[6] David J. C. Mackay. Good error correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory, no. 2*, 45(2):399–431, March 1999.

[7] Wolfgang Proß. *Design of Robust 2D Barcodes for Industrial Environments*. PhD thesis, Karlsuhe University, 2012.

[8] Robert Michael Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547, 1981.

[9] D.C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up*. SPRINGER, 2010.

[10] Raul Camposano. From behavior to structure: High level synthesis. *IEEE Transactions on Signal Processing*, pages 8–19, October 1990.

[11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 30(4), April 2011.

[12] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Des Autom Embed Syst*, August 2012.

[13] David J. Pursley and Brett L. Cline. A practical approach to hardware and software soc tradeoffs using high-level synthesis for architectural exploration. *Proc. Global Telecommunications Conf.*, April 2003.

[14] A. C. Carusone, A. Darabiha, and F. R. Kschischang. A bit-serial approximate min-sum ldpc decoder and fpga implementation. *IEEE Int. Symp. Circuits and Sys-tems, ISCAS*, May 2006.

[15] S. G. Wilson, R. Zarubica, and E. Hall. Multi-gbps fpga-based low-density parity-check (ldpc) decoder design. *Proc. Global Telecommunications Conf.*, pages 548–552, November 2007.

[16] Saeed Sharifi Tehrani, Shie Mannor, and Warren J. Gross. Fully parallel stochastic ldpc decoders. *IEEE Transactions on Signal Processing*, 56(11):1–12.

[17] Z. Wang and Z. Cui. Low-complexity high-speed decoder design for quasi-cyclic ldpc codes. *IEEE Trans. VLSI Syst.*, 15(1):104–114, 2007.

[18] X. Chen, J. Kang, S. Lin, and V. Akella. Memory system optimization for fpga-based implementation of quasi-cyclic ldpc codes decoders. *IEEE Trans. Circuits Syst. I, Reg. Papers*, 58(1):98–111, January 2011.

[19] Vikram Arkalgud Chandrasetty and Syed Mahfuzul Aziz. Fpga implementation of a ldpc decoder using a reduced complexity message passing algorithm. *JOURNAL OF NETWORKS*, 6(1):36–45, 2011.

[20] JIN SHA, MINGLUN GAO, ZHONGJIN ZHANG, LI LI, and ZHONGFENG WANG. A memory efficient fpga implementation of quasi-cyclic ldpc decoder. *Proceedings of the 5th WSEAS Int. Conf. on Instrumentation*, pages 218–223, 2006.

[21] Mohammad M. Mansour and Naresh R. Shanbhag. Low power vlsi decoder architectures for ldpc codes. *Proceedings of the 5th WSEAS Int. Conf. on Instrumentation*, pages 284–289, August 2002.

[22] Hao Zhong and Tong Zhang. Design of vlsi implementation-oriented ldpc codes. *Proc. IEEE Semiann. Vehicular Technology Conf.*, pages 670–673, October 2003.

[23] S. Mannor, S. S. Tehrani, and W.J. Gross. Fully parallel stochastic ldpc decoders. *IEEE Trans. Sig. Proc.*, 56(11):5692–5703, 2008.

[24] Ali Naderi, Shie Mannor, Mohamad Sawan, and Warren J. Gross. Delayed stochastic decoding of ldpc codes. *IEEE TRANSACTIONS ON SIGNAL PROCESSING*, 59:1–25, 2008.

[25] Alexios Balatsoukas-Stimming and Apostolos Dollas. Fpga-based design and implementation of a multi-gbps ldpc decoder. *Field Programmable Logic and Applications*, pages 262–269, August 2012.

APPENDIX A RTL ALLOCATION REPORT

Allocation Report for all threads:

	Per Instance			Total	
Resource	Count	LUTs	Mults	LUTs	Mults
dut_Mux62_2	3	62	0	186	0
dut_Mux16_2	8	16	0	128	0
dut_MuxS62_4	1	124	0	124	0
dut_Mux17_14	1	119	0	119	0
dut_Mux9_6	3	27	0	81	0
dut_Mux16_8	1	64	0	64	0
dut_Sub_62S_4	1	63	0	63	0
dut_Add_62S_3	1	62	0	62	0
dut_Mux62_3	1	62	0	62	0
dut_MuxS62_2	1	62	0	62	0
dut_Mux17_2	3	17	0	51	0
dut_Minus_62S_0	1	46	0	46	0
dut_Mux9_4	2	18	0	36	0
Eq16	6	6	0	36	0
Add17	2	18	0	36	0
dut_Mux17_4	1	34	0	34	0
dut_Mux16_3	2	16	0	32	0

dut_GreaterThanEQ_1U_5	1	31	0	31	0
dut_LessThan_1U_1	1	31	0	31	0
dut_GreaterThan_1U_2	1	31	0	31	0
dut_Mux12_4	1	24	0	24	0
dut_Mux9_5	1	23	0	23	0
dut_Mux8_5	1	20	0	20	0
dut_Mux9_2	2	9	0	18	0
dut_Mux9_3	2	9	0	18	0
Add16	1	17	0	17	0
dut_Mux17_3	1	17	0	17	0
Sub16	1	17	0	17	0
GtLt8	2	8	0	16	0
GtLt16	1	16	0	16	0
dut_Mux8_4	1	16	0	16	0
dut_Mux1_3	15	1	0	15	0
dut_Mux12_2	1	12	0	12	0
Add8	1	9	0	9	0
GtLt9	1	9	0	9	0
Xor8	1	8	0	8	0
dut_Mux8_2	1	8	0	8	0
Ne11	1	4	0	4	0
Eq12	1	4	0	4	0
RedOr8	1	3	0	3	0
And_1	3	1	0	3	0
Xor2	1	2	0	2	0
dut_Mux1_2	2	1	0	2	0
dut_gen_busy	1	2	0	2	0
RedOr2	1	1	0	1	0
Xor1	1	1	0	1	0
Or_1	1	1	0	1	0
Ne2	1	1	0	1	0

RedNor1	2	0	0	0	0
dut_RAM_63504X62_5	1	-1	0	-1	0
dut_RAM_4096X16_2	1	-1	0	-1	0
dut_RAM_63504X16_4	2	-1	0	-2	0
dut_RAM_504X62_3	2	-1	0	-2	0
dut_RAM_127008X62_6	3	-1	0	-3	0
dut_RAM_504X16_1	6	-1	0	-6	0
MulS_dedicated9	1	0	1	0	1
MulS_dedicated10	1	0	1	0	1
Register bits	1634	0	0	0	0
Implicit mux LUTs				945	
Estimated cntrl				1188	
Total LUTs/Mults				3735	2

APPENDIX B
LOGIC SYNTHESIS RESOURCE USAGE REPORT

Resource Usage Report for dut

Mapping to part: xc6vhx565tff1923-1

Cell	Usage
DSP48E1	2
FD	103
FDE	1249
FDR	4
FDRE	113
FDS	1
FDSE	2
GND	75
MUXCY	28
MUXCY_L	305
MUXF7	11
RAMB36E1	524
VCC	75
XORCY	198
LUT1	12
LUT2	278
LUT3	375

LUT4	264
LUT5	705
LUT6	899
LUT6_2	5
I/O ports:	70
I/O primitives:	54
IBUF	19
IBUFG	1
OBUF	34
BUFG	1
I/O Register bits:	25
Register bits not including I/Os:	1447
RAM/ROM usage summary	
Block Rams	524 of 912
DSP48s:	2 of 864
Global Clock Buffers:	1 of 32
Total load per clock:	
dut|clk	2002
Mapping Summary	
Total LUTs:	2135
Unique control sets:	72

APPENDIX C
PLACE AND ROUTE DEVICE UTILIZATION SUMMARY

Device Utilization Summary	
Slice Logic Utilization:	
Slice Registers:	1,447 out of 708,480
used as Flip Flops	1447
used as Latches	0
used as Latch-thrus	0
used as AND/OR logics	0
Slice LUTs	1,977 out of 354,240
used as logic	1,956 out of 354,240
using O6 output only	1325
using O5 output only	6
using O5 and O6	625
used as ROM	0
used as Memory	0 out of 101,920
used exclusively as route-thrus	21
with same-slice register load	19
with same-slice carry load	2
with other load	0

Slice Logic Distribution	
Occupied Slices	805 out of 88,560
LUT Flip Flop pairs used	2577
with an unused Flip Flop	1,161 out of 2,577
with an unused LUT	600 out of 2,577
Fully used LUT-FF pairs	816 out of 2,577
Slice register sites lost	
to control set restrictions	0 out of 708,480
IO Utilization	
bonded IOBs	54 out of 720
IOB Flip Flops	25
Specific Feature Utilization	
RAMB36E1/FIFO36E1s	524 out of 912
using RAMB36E1 only	524
using FIFO36E1 only	0
RAMB18E1/FIFO18E1s	0 out of 1,824
BUFG/BUFGCTRLs	1 out of 32
used as BUFGs	1
used as BUFGCTRLs	0
ILOGICE1/ISERDESE1s	13 out of 720
used as ILOGICE1s	13
used as ISERDESE1s	0
OLOGICE1/OSERDESE1s	12 out of 720
used as OLOGICE1s	12
used as OSERDESE1s	0
DSP48E1s	2 out of 864
STARTUPs	1 out of 1

APPENDIX D PROJECT.TCL

```
#
# Cynthesizer Project File
#
# This file contains all of the basic options needed to synthesize
# and simulation a Cynthesizer project
#

#
# Technology Libraries
#

set LIB_PATH "$env(FORTE_HOME)/examples/tutorials"

techLib          "$LIB_PATH/exampleLib_5000ps/sxlib013.lib"
techLib          "$LIB_PATH/exampleLib_5000ps/dff013.lib"

cynthLib         "$LIB_PATH/exampleLib_5000ps" -optional
cynthLib         "./NOVA_Cynw_Virtex6_XC6VHX565T_FF1923_1_25ns" ↩
   -optional

#
#In FPGA the default is ns, in ASIC the default is ps
# Basic GCC compiler options
#
ccOptions       "-DCLOCK_PERIOD=25"

#
```

```
# Global Command Line Options
#
cynthHLOptions "--clock_period=25"
cynthHLOptions "--inline_partial_constants=on"

cynthHLOptions "--timing_aggression=-10.0"  ##Helps with timing

#cynthHLOptions "--sched_asap=on"       ##timing problems
#cynthHLOptions "--balance_expr=delay"      ##timing problems
#cynthHLOptions "--unroll_loops=on"         ##design gets too larg
#cynthHLOptions "--flatten_arrays=all_const"    ##timing problems


# Simulation Options
#

verilogSimulator mti
endOfSimCommand  "\@make --no-print-directory compare"

#
# Testbench or System Level Modules
#
systemModule main.cc system.cc tb.cc

###################################################################
# Synthesis Module Configurations - SC_MODULES to be Cynthesized
###################################################################

# The cynthConfigs use the [lib ...] subcommand to select a
# specific cynthLib to use. The RTL will be optimized for the ↩
   specific
# FPGA device characterized in the library.
cynthModule dut dut.cc \
    [cynthConfigs {
        BASIC [lib exampleLib_5000ps]
    V2P [lib NOVA_Cynw_Virtex6_XC6VHX565T_FF1923_1_25ns] ↩
   --inline_partial_constants=on
    }]
```

```
####################################################################
# Logic Synthesis Configurations
####################################################################

# For FPGA targets, SynplifyPro is used as the logic synthesis
# command.

logicSynthCommand bdw_runspro

# The logicSynthConfigs specify a cynthConfig as input; the RTL
# generated for that cynthConfig will be given to
# SynplifyPro. Xilinx-specific option settings are passed via an
# options file (spro_xilinx_options.tcl) located in this
# directory. Please see that file for details.

logicSynthConfig LS_V2P {dut V2P} \
    [options {BDW_LS_OPTION_FILE spro_xilinx_options.tcl}]

####################################################################
# P&R Configurations
####################################################################

# For each FPGA, the appropriate tool chain is used for P&R.

# The pnrConfigs specify a logicSynthConfig as input. The gate-level
# Verilog generated by SynplifyPro is used as input to the P&R tools.

pnrConfig PNR_V2P LS_V2P [pnrCmd bdw_run_xilinx]

#
# Simulation Configurations
# for ASIC
simConfig BEH     { dut BEH   BASIC } [ escArgv "" ]
simConfig BASIC_C { dut RTL_C BASIC } [ escArgv "" ]
simConfig BASIC_V { dut RTL_V BASIC } [ escArgv "" ]

####################################################################
# Simulation Configurations
# for FPGA
####################################################################

# Behavioral simulation

simConfig B {dut BEH}

# Behavioral simulation using the V2P results

simConfig B_V2P {dut BEH V2P}

# RTL SystemC simulations, one for each cynthConfig

simConfig C_V2P {dut RTL_C V2P}

# RTL Verilog simulations, one for each cynthConfig
# We need the appropriate Verilog primitive libraries for each

simConfig V_V2P {dut RTL_V V2P}

# Check to see if we have the appropriate tools/libraries available.

set VERBOSE 0

if { [array names env -exact XILINX] == "" } {
    if { $VERBOSE } {
    puts stderr ""
    puts stderr "Warning: The \$XILINX environment variable is not ↩
   set."
    puts stderr "Check Xilinx toolset installation, needed for ↩
   Verilog simulation libraries."
    puts stderr "Using the current directory as the source of ↩
   libraries."
    puts stderr "Gate-level simulations will not work if libraries ↩
   are not available."
    puts stderr ""
    }
    set XIL_DIR "." ;# probably will not work
} else {
    set XIL_DIR $env(XILINX)
}

# Gate-level Verilog (pre-P&R, untimed) simulation, one for each
# logicSynthConfig. Note the special Verilog libraries required for
# simulations. The [nosdf] command is required for pre-P&R ↩
   simulations
# since SynplifyPro does not produce an SDF file.

simConfig VG_V2P {dut GATES_V LS_V2P} \
    [vlogFiles $XIL_DIR/verilog/src/glbl.v] \
    [vlogTopModules glbl] \
    [vlogLibs $XIL_DIR/verilog/src/unisims] \
    [nosdf]



# Gate-level Verilog (post-P&R, timed) simulation, one for each
# pnrConfig. Since the P&R tools produce an SDF file, it will be used
# in the simulation for full timing analysis.

simConfig VPNR_V2P {dut GATES_V PNR_V2P} \
    [vlogFiles $XIL_DIR/verilog/src/glbl.v] \
    [vlogTopModules glbl] \
    [vlogLibs $XIL_DIR/verilog/src/simprims]
```