DIPLOMA THESIS

Testbed Implementation for a Low Density Parity Check Decoder

Institute of Communications and Radio-Frequency Engineering
Vienna University of Technology (TU Wien)

Telecommunications Research Center Vienna (ftw.)

Supervisors:
Univ. Prof. Dipl.-Ing. Dr. techn. Markus Rupp (TU Wien)
Dipl.-Ing. Gottfried Lechner (ftw.)

Christoph Angerer
Neustiftgasse 12/1/7, 1070 Vienna
Register No. 9925554

Vienna, 13th May 2005 _____

DIPLOMARBEIT

Aufbau und Inbetriebnahme eines Testbeds für einen Low Density Parity Check Dekoder

Institut für Nachrichtentechnik und Hochfrequenztechnik
Technische Universität Wien (TU Wien)

Forschungszentrum Telekommunikation Wien (ftw.)

Betreuer:
Univ. Prof. Dipl.-Ing. Dr. techn. Markus Rupp (TU Wien)
Dipl.-Ing. Gottfried Lechner (ftw.)

Christoph Angerer
Neustiftgasse 12/1/7, 1070 Wien
Matrikelnummer 9925554

Wien, 13. Mai 2005 _____

Abstract

During the recent years lots of theoretical work has been done on decoding Low Density Parity Check (LDPC) codes iteratively. By doing so, very good error correction performance at transmission rates very close to channel capacity can be achieved, at a decoding complexity that is linear in the block length.

However these theoretical results can not be directly applied to practical systems, as the complexity of the decoding algorithm is too high and finite precision, finite block lengths and a finite amount of resources limit the decoder performance. Moreover LDPC codes are usually randomly constructed for theoretical considerations, which has the practical disadvantage that the entire decoder matrix has to be stored. In addition to that, scheduling becomes a major problem which perhaps is impossible to solve when using parallel structures. Therefore, deterministic code constructions which structure the code may become attractive. By constructing a code the problems of scheduling can be addressed, moreover only a few parameters need to be stored in order to derive the parity check matrix. On the other hand, deterministic construction and too much regularity in the parity check matrix weaken the error correcting performance of the code. Hence, some trade offs between decoder performance and realizability have to be accepted in practical implementations.

The objective of this work is to implement a scalable LDPC decoder, meaning that the design can easily be reconfigured in order to derive different decoders regarding their parameters, such as bus widths, number of parallel processing units and code parameters. The emphasis lies on scheduling in parallel architectures, partly deterministic code construction which compromises decoder performance on the one hand and mapping to parallel architectures and deterministic derivation of the parity check matrix on the other hand. Also automatic VHDL module generation and component instantiation is addressed. By defining some code parameters, a code can be constructed and synthesizeable VHDL modules, which decode this code, can be generated.

Kurzfassung

Während der letzten Jahre wurde im Bereich iterative Dekodierung von Low Density Parity Check (LDPC) Kodes viel geforscht. Dadurch kann eine sehr gute Fehlerkorrektur bei Übertragungsraten nahe an der Kanalkapazitätsgrenze erreicht werden, wobei die Dekodierkomplexität jedoch nur linear mit der Blocklänge des Kodes steigt.

Diese theoretischen Ergebnisse finden in der Praxis jedoch oft keine direkte Anwendung, weil die Komplexität des Dekodieralgorithmus zu hoch ist, und endliche Wortbreiten, endliche Blocklängen, und beschränkte Resourcen die Dekodierungsleistung einschränken. Ausserdem werden LDPC Kodes für theoretische Untersuchungen meist mit zufälligen Verfahren konstruiert, was den Nachteil mit sich führt, dass die gesamte Parity Check Matrix gespeichert werden muss, und Scheduling bei parallelen Architekturen zu einem unlösbaren Problem werden kann. Deshalb werden bei praktischen Anwendungen deterministisch konstruierte Kodes interessant. Konstruiert man einen Kode, so kann auf Scheduling Rücksicht genommen werden, und meist genügen wenige Parameter um die Parity Check Matrix ableiten zu können. Andererseits wird durch zu viel Regularität in der parity check Matrix die Fehlerkorrekturleistung des Kodes geschwächt. Deshalb muss in praktischen Systemen ein Kompromiss zwischen Realisierbarkeit und Dekodierleistung eingegangen werden.

Das Ziel dieser Arbeit ist, einen skalierbaren LDPC Dekoder zu implementieren, wobei man das Design durch Einstellen weniger Parameter, wie Busbreiten, Kode Parameter oder Anzahl der parallelen Einheiten, leicht umkonfigurieren können soll. Der Schwerpunkt dieser Arbeit liegt beim Scheduling in parallelen Architekturen, und der deterministischer Kodekonstruktion, welche Kompromisse zwischen Dekodierungsleistung einerseits und Abbilden des Kodes auf parallele Architekturen und deterministische Ableitung der Parity Check Matrix andererseits eingeht. Ausserdem wird automatische Generation von VHDL Modulen, die zu dem konstruierten Kode passen, angesprochen. Automatische Erzeugung des Kodes und automatische Konfiguration der Module und einer passenden Architektur sollen erreicht werden.

Contents

1 Introduction

1.1 Low Density Parity Check Codes

Low density parity check codes (LDPC codes) were first invented by Gallager in 1962 [1]. First nobody paid too much attention to them until they were rediscovered by MacKay in 1999 [2]. LDPC codes belong to the class of channel codes, which are used to correct errors caused by noise in a digital transmission system. Figure 1.1 shows a digital transmission system, consisting of a data source, a channel encoder, a transmitter, the channel, a receiver, a channel decoder and finally the data sink. As LDPC codes are linear codes, the errors caused by the noise, can be detected and corrected by means of additional parity bits. In order to do this, the channel encoder awaits a block of data u (length K), produces the code words x (length N) by multiplying u with the encoder Matrix G and hands the code word to the transmitter. Thereby, the encoder introduces some redundancy, which can later be used at the decoder to detect and correct transmission errors. The code rate R is defined as $R = K/N$. The transmitter maps the digital data to a transmit signal t which then is transmitted over the channel. The channel introduces some distortions, modeled by the additive noise n. The receiver observes the incoming signal r, transforms it to a block of soft decisions y, and hands it to the decoder. If these soft decisions are used to produce hard decisions $\hat{x}$, they usually differ from x due to the noise added at the channel. By means of the parity bits introduced by the encoder, the decoder now can estimate the transmitted data. Any vector $\hat{x}$ is a code word, if $H \cdot \hat{x} = 0$, with H denoting the parity check matrix of size $M \times N$, with $M = N - K$. If the channel is

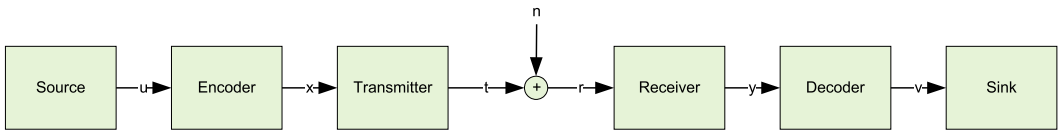

Figure 1.1: Digital Communication System.

not considered, $y = G \cdot u$, $H \cdot G \cdot u = 0$, and further $H \cdot G = 0$, for all linear codes.

As long as communicating at code rates below channel capacity, any arbitrary reliability can be achieved, whereas it is impossible to achieve arbitrary reliability at code rates above channel capacity (Shannon's channel coding theorem).

In this work I am only focusing on the channel decoder when using LDPC codes.

1.1.1 Decoding LDPC codes

As already mentioned, the decoder of an LDPC code is described by its parity check matrix H, which can be equivalently represented by a factor graph (see Fig. 1.2). In the factor graph, every variable node (circles on the left hand side) corresponds to a column in the parity check matrix, and every check node (squares on the right hand side) corresponds to a row in the parity check matrix. Hence, there are M check nodes and N variable nodes. The connecting edges between the variable nodes and the check nodes correspond to the nonzero positions in the parity check matrix.

For example, the first row in the parity check matrix is represented by check node A (CN A), the first column by variable node 1 (VN 1). Edges connect VN 1 with CN D and CN F, according to the '1s' in the first column (in row 4 and row 6). CN A is connected to VN 2, VN 3, VN4 and VN 12, according to the '1s' in the first row at the corresponding positions.

The weight of a row or the check node degree d_c is the number of nonzero entries in this row, the weight of a column or the variable node degree d_v is the number of '1s' in this column. Regular LDPC codes have the same variable node degree for all VNs and the same check node degree for all CNs, whereas irregular LDPC codes can have arbitrary weight distribution. For regular codes, the total number of edges connecting check nodes and variable nodes is then $d_v \cdot N = d_c \cdot M$. In this work I am only considering regular LDPC codes. Regular LDPC codes are often denoted by (d_v, d_c) LDPC code of length N, the example in Fig. 1.2 would be a (2,4) LDPC code of length 16.

Sum - Product Algorithm

The decoding algorithm, known as the Sum - Product Algorithm (SPA) or Belief Propagation (BP) Algorithm is an iterative algorithm [2]. Messages from

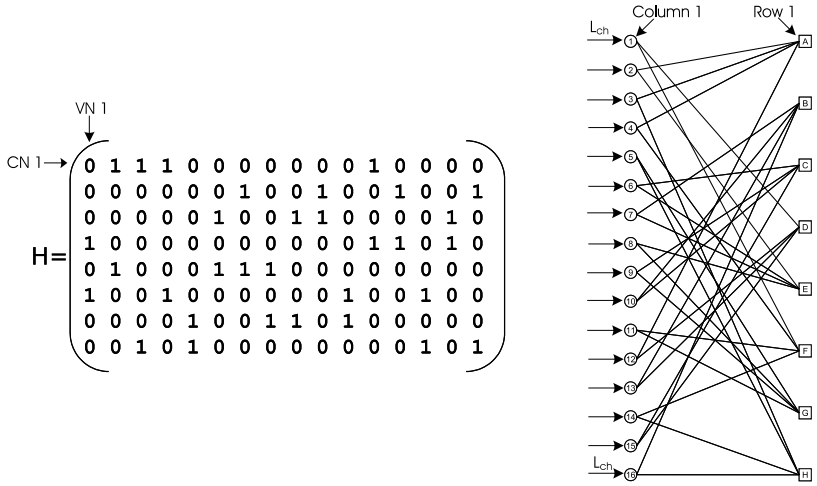

Figure 1.2: Parity Check Matrix and Factor Graph.

the channel are handed to the variable nodes, which process some operations on these messages and the messages from the check nodes, and send the results along the edges in the factor graph to the check nodes (see Fig. 1.2). The check nodes take these messages, also compute some operations, and send the results back to the variable nodes. These two operations are processed alternately until a maximum number of iterations is reached. The channel values and the messages that are sent between variable nodes and check nodes are Log Likelihood Ratios (LLRs). The variable nodes receive different observations of the same random variable. If these observations are independent, we can sum the LLRs of these observations, which is done in the variable nodes. The LLR of a random variable, which is the XOR of other random variables can be calculated by using the tanh-rule (see Equ. (1.2)). This operation is computed in the check nodes. Thus, variable and check nodes compute the result that is sent along one edge to a check node by performing an operation on all the other incoming messages, and the check nodes also compute an outgoing message as a function of all the other incoming messages (see Fig. 1.3). These operations are:

$$\text{variable nodes:} \qquad L_i = L_{ch} + \sum_{j \neq i} L_j, \tag{1.1}$$

$$\text{check nodes:} \qquad L_i = 2 \tanh^{-1} \left(\prod_{j \neq i} \tanh \frac{L_j}{2} \right). \tag{1.2}$$

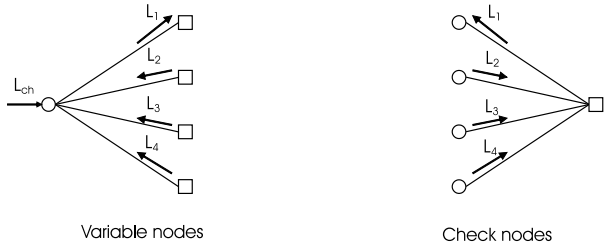

Figure 1.3: Variable node and check node operations.

A derivation of the algorithm can be found in [2]. Note that independence of the observations is assumed, which means that the factor graph is cycle free. This is never satisfied in practice, but the algorithm has good performance if the cycles are long enough.

An important feature of LDPC codes is that the decoding complexity is *linear* in the block length, when decoding with the Sum - Product algorithm. This is very important as LDPC codes can achieve rates arbitrarily close to channel capacity if the block length is large. In a practical system, block lengths between 1000 to 10000 bits should be considered. On the other hand, it is very hard to implement the check node operation (Equ. (1.2)) in hardware, as it consists of a product and the two nonlinear functions tanh and $\tanh^{-1}$. If it was directly implemented, the tanh and $\tanh^{-1}$ would need to be implemented as a look up table and a product would have to be computed. The variable node operation (Equ. (1.1)) however, can easily be implemented on a chip. Hence, the check node operation is often approximated by a simpler operation (see next section). The resulting algorithm is then known as the Sum - Min algorithm (SMA), which is usually implemented in practical systems.

Sum - Min Algorithm

In this algorithm, the check node operation (Equ. (1.2)) is approximated by a simpler operation, whereas the variable node operation (Equ. (1.1)) remains the same. Recall the check node equation,

$$L_i = 2\tanh^{-1}\left(\prod_{j\neq i}\tanh\frac{L_j}{2}\right),$$

which can be approximated by

$$L_i \;=\; 2\tanh^{-1}\prod_{j\neq i}\left(\operatorname{sign}(L_j)\tanh\frac{|L_j|}{2}\right)$$

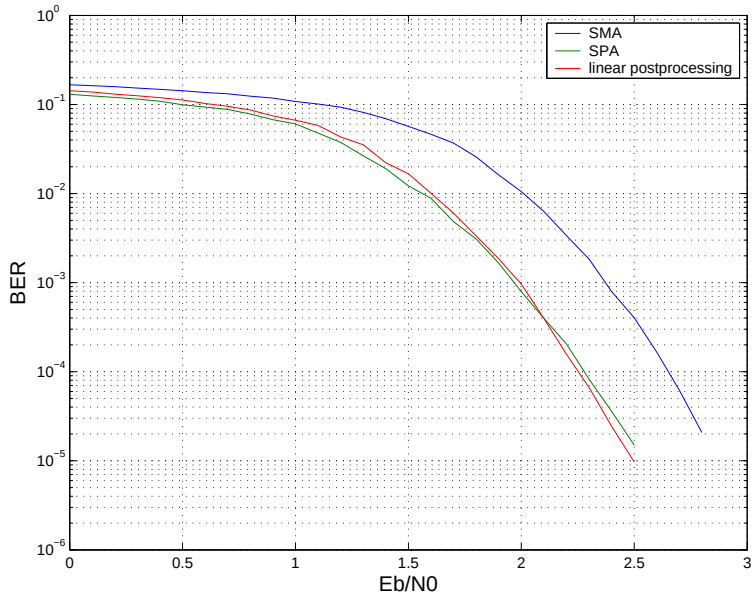

Figure 1.4: BER for SMA, SPA and linear postprocessing for a $(3, 6)$ length 1024 code.

$$\begin{aligned} &\approx 2\tanh^{-1}\left(\min_{j\neq i}\left(\tanh\frac{|L_j|}{2}\right)\prod_{j\neq i}\operatorname{sign}(L_j)\right) \\ &= 2\prod_{j\neq i}\operatorname{sign}(L_j)\tanh^{-1}\left(\tanh\min_{j\neq i}\left(\frac{|L_j|}{2}\right)\right) \\ &= \min_{j\neq i}|L_j|\prod_{j\neq i}\operatorname{sign}(L_j). \end{aligned} \tag{1.3}$$

This function now can easily be realized on hardware, as it is only an XOR operation of the signs multiplied by the smallest incoming absolute value. Due to this approximation, the decoding performance decreases, as shown in Fig. 1.4, but by applying a post processing function to the outgoing check node messages, the performance can be increased again.

Postprocessing of Check Node Messages

As shown in [3], the true LLR of an outgoing check node message is a function of the mutual apriori information between the check node inputs and the code digits (I_{AC}), the check node degree (d_c) and the message computed by the Sum - Min algorithm (L^{SMA})

$$L = f(d_c, I_{AC}, L^{SMA}). \tag{1.4}$$

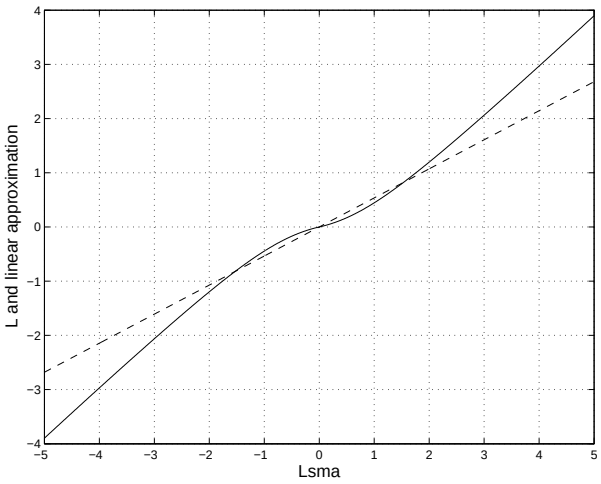

Figure 1.5: Postprocessing function for $d_c = 6$, $I_{AC} = 0,5$ and linear approximation.

Figure 1.5 shows this function (solid) for an AWGN extrinsic channel, with $I_{AC} = 0,5$ and $d_c = 6$. As Equ. (1.4) is again a nonlinear function and therefore difficult to implement, it is again approximated, either by a scaling or an offset function [4]. The dashed line in Fig. 1.5 is the linear approximation:

$$L = \frac{L^{SMA}}{\alpha}. \tag{1.5}$$

The approximated LLR using the offset function β is:

$$L = \text{sign}(L^{SMA}) \cdot \max(|L^{SMA}| - \beta, 0). \tag{1.6}$$

The factors α and β are calculated by minimizing the expected squared error and depend on d_c and I_{AC}:

$$\alpha = \arg\min_{\acute{\alpha}} \int_{-\infty}^{\infty} \left[f(d_c, I_{AC}, L^{SMA}) - \frac{1}{\acute{\alpha}} L^{SMA} \right]^2 \cdot p(L^{SMA}, d_c) dL^{SMA} \tag{1.7}$$

$$\beta = \arg\min_{\acute{\beta}} \int_{-\infty}^{\infty} \left[f(d_c, I_{AC}, L^{SMA}) - \text{sign}(L^{SMA}) \cdot \max(|L^{SMA}| - \acute{\beta}, 0) \right]^2 \cdot \cdot p(L^{SMA}, d_c) dL^{SMA}. \tag{1.8}$$

Note that the operations of the Sum - Min algorithm are homogeneous operations, meaning that a scaling of the channel values results in a scaled version of the results. This property holds true, if the linear postprocessing is applied, but is destroyed if the offset function is applied. Hence, if this property is desired, the linear function has to be chosen. On the other hand, due to

finite precision, only certain values of $1/\alpha$ can be chosen in a practical system, for example 0,75 or 0,875. The improvement of the SMA by applying postprocessing can clearly be seen in Fig. 1.4.

Quantization

Using less bits for representing the internal messages leads to a decoding performance loss, as shown in the simulation of Fig. 1.6. The simulation was done for a (3,6) length 1024 code, using the Sum - Product algorithm and 100 iterations for decoding. For the internal representation of the messages floating point numbers, 8 bit, 6 bit and 4 bit numbers (two's complement) were used. Moreover the decoding perfomance for the same code, using 8 bits and 20 iterations is included in the figure. The quantization intervals $0,1$, $0,3$, and $0,7$ were chosen for the 8 bit, 6 bit and 4 bit representation respectively. Note that the decoding performance is basically the same for using floating point numbers and 8 bit fix point numbers, and only slightly differs for using quantized messages with 6 bits. Even an internal representation of 4 bits could be sufficient for many applications. If the quantization interval was chosen too small, all values would be in saturation after a few iterations, if it was chosen too large, the quantization noise of the channel messages would be quite high.

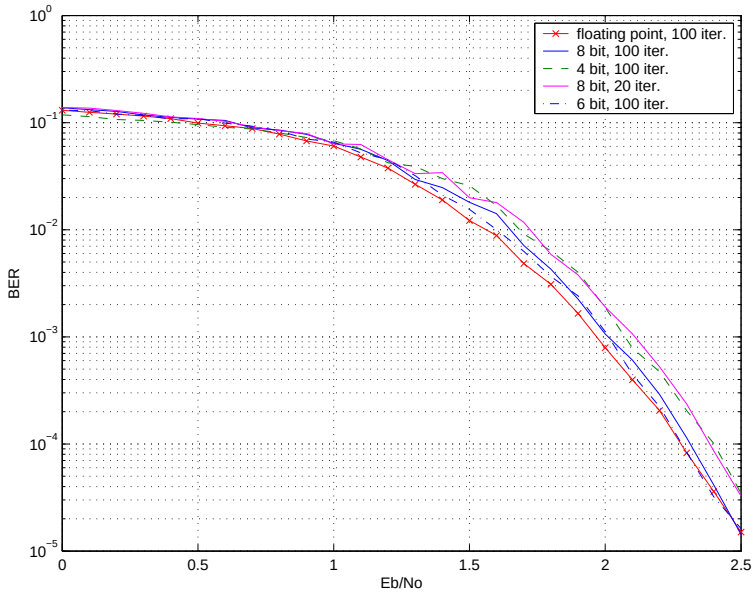

Figure 1.6: Decoding performance using different internal bit widths.

1.2 Field Programmable Gate Array

For this work I chose a field programmable gate array (FPGA) as a platform to realize an LDPC decoder. I chose a Spartan 3 FPGA from Xilinx, placed on a development board of Digilent. A USB interface between the Spartan 3 Board and the PC is used, which allows testing and verification of the architecture by transferring generated data between the PC and the FPGA architecture. The FPGA was programmed and configured with VHDL and the Xilinx integrated software environment (ISE).

1.2.1 Spartan 3 Board

The Spartan 3 Board provided by Digilent is a low cost development board and evaluation platform for Spartan 3 FPGA designs [10]. The board with its main components is shown in Fig. 1.7. Number 1 in Fig. 1.7 indicates the XC3S400 FPGA [9](see Section 1.2.2), number 2 shows a 2Mbit Xilinx XCF02S platform flash, used for storing the FPGA configuration. Number 3 are expansion connectors for connecting additional daughter cards, which I used for attaching the USB daughter card which enables communication with the PC. Number 4 indicates the JTAG interface for configuring the FPGA and the PROM module. Several different configuration modes can be selected by positioning the jumpers (number 5) on the Spartan 3 Board. Moreover some additional ports, a 7-segment display, buttons, LEDs and switches can be seen in Fig. 1.7 which can be used for debugging for example, but are not used in my design. On the bottom of the board, an additional 1 Mbyte SRAM and a 50MHz crystal clock are located. The FPGA pins can be assigned using Xilinx PACE software, a schematic of the board and its connections to the FPGA is shown in [10].

1.2.2 Spartan 3 FPGA

Spartan 3 FPGAs are low cost FPGAs. All Spartan 3 FPGAs are supported by the Xilinx ISE for programming. For this work I used an XC3S400 device with the following resources:

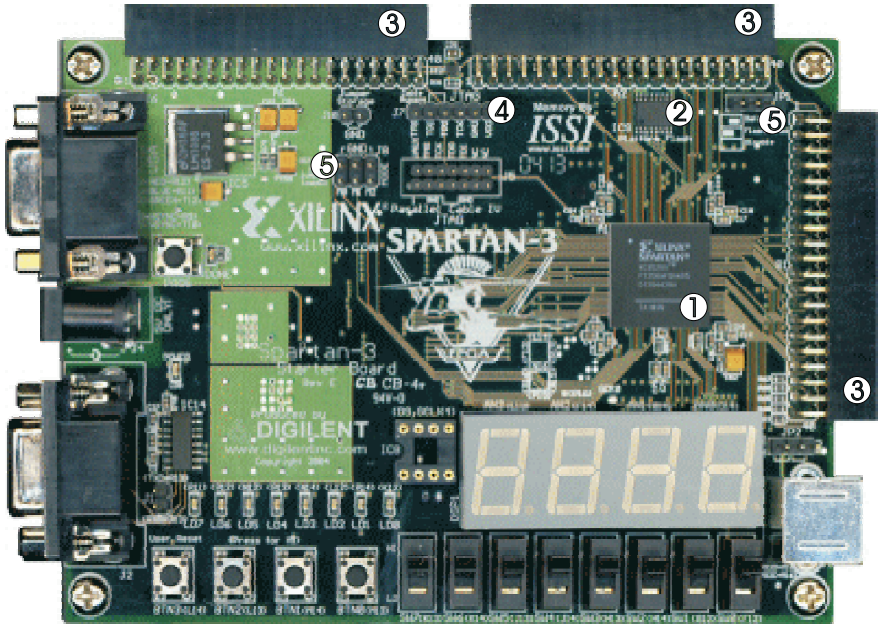

Figure 1.7: Spartan 3 Board.

system gates:	400K
equivalent logic cells:	8064
configurable logic blocks:	896
distributed RAM:	56Kbits
block RAM:	288Kbits
dedicated multipliers:	16
digital clock managers:	4

The Spartan 3 family architecture consists of five fundamental programmable logic elements:

- configurable logic blocks (CLBs): contain RAM based look up tables (LUT) to implement logic and storage elements.
- input output blocks (I/O): control the flow of data between the I/O pins and the internal logic of the device.
- block RAM: provides data storage in the form of 18-Kbit dual-port blocks.
- multipliers: accept two 18bit inputs and calculate the product.
- Digital Clock Manager (DCM): provide solutions for clock manipulation such as delaying, multiplying, phase shifting and dividing of clock signals.

Spartan-3 FPGAs are programmed by loading configuration data into robust static memory cells that collectively control all functional elements and routing resources. Before powering on the FPGA, configuration data is stored externally in the PROM. After applying power, the configuration data is written to the FPGA.

Configurable Logic Blocks

The Configurable Logic Blocks (CLBs) constitute the main logic resource. Each CLB comprises four interconnected slices. All four slices have the following elements in common: two logic function generators, two storage elements, multiplexers, carry logic, and arithmetic gates. The slices use these elements to provide logic, arithmetic, and ROM functions. Besides these, the two of the slices support two additional functions: storing data using distributed RAM and shifting data with 16-bit registers. The RAM-based function generator, also known as a look up table or LUT, is the main resource for implementing logic functions. The multiplexers are used to combine LUTs in order to permit more complex functions. Dedicated logic gates as XORs and ANDs support the implementation of mathematical operations.

Block RAM

The on chip block RAM is organized as configurable, synchronous 18Kbit blocks. Block RAM stores relatively large amounts of data more efficiently than the distributed RAM feature described earlier. The latter is better suited for buffering small amounts of data anywhere along signal paths. The width and depth of each block RAM is configurable up to a certain extent, a width of 1, 2, 4, 8, 16 or 32 bits can be chosen. The number of addressable locations is then given by the total RAM size. Furthermore, multiple blocks can be cascaded to create still wider and deeper memories.

The block RAM has a dual port structure. The two identical data ports called port A and port B permit independent access to the common RAM block, which has a maximum capacity of 16,384 bits. The remaining 2048 bits to 18Kbit can be used as parity bits. Each port has its own dedicated set of data, control and clock lines for synchronous read and write operations. In my design all block RAMs are configured as dual port memories.

1.2.3 Communication Interface

Data transfer and JTAG programming are offered by the Digilent USB 2 daughter card. This enabled me to test and verify the design running on the FPGA by sending test data generated in Matlab to the chip and reading the results.

USB interface

The Digilent USB 2 module [11, 12] provides a prepared USB port that can add USB 2.0 connectivity to the Spartan 3 system board, by attaching it at one of the expansion connectors. The interface between USB board and Spartan 3 system board is a parallel one (see Fig. 1.8). The USB port can be used to configure the FPGA on an attached system board, and it can also be used for general data transfers. User data transfer works byte-wise. The USB 2 board is based on a Cypress CY7C68013 device. Firmware in the Cypress chip works with Digilent's PC-based Adept software to coordinate JTAG programming and user data transfers. For user data transfers, Digilent provides a DLL, API and the required Windows drivers. This allows creating applications that can communicate with the FPGA via the USB module.

C++, Matlab Routines

The DPCUTIL Dynamic Link Library (DLL) [13] provides an Applications Programming Interface (API) that allows applications software running under Microsoft Windows on a host computer to communicate with Digilent system boards. The DPCUTIL DLL works with the USB 2 board and provides the communication channel between the host PC and the attached system board. The DPCUTIL data transfer functions require that the gate array configuration contains a parallel port interface (see Fig. 1.8) based on and compatible with the Digilent Parallel Interface Module specification and reference design (this is described in Section 2.1). This interface provides a mechanism for the user to define a set of addressable registers on the FPGA that can be

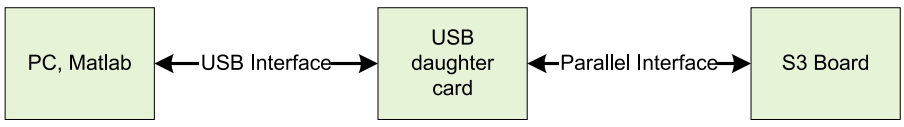

Figure 1.8: USB Interface.

written to or read from by the DPCUTIL data transfer API functions. The data transfer API functions allow writing or reading a single register, writing or reading sets of registers, or reading or writing a stream of data into or out of a single register. The latter two functions are the ones used in my design. The API is defined as a set of C callable functions, and can be used with programs written in either C or C++. In order to access the API, I created a Matlab DLL (mex function).

This mex function (called decode(y), with y denoting a block to transmit) first initializes and opens the connection to the FPGA, then it streams an entire block from a specified register on the FPGA to the host PC. Next the block y is transmitted to another dedicated register on the FPGA. Finally the connection is closed and the read block is returned. During streaming I monitored the transfer rate, which is 12 cycles/transfer for writing and reading (during one transfer one byte is transmitted), which leads to 24 cycles per transfer at a clock frequency of 50MHz. This would then give a transfer rate of approximately 2Mbyte/s. But in between reading and writing an entire block, quite a big gap can be seen (measured with ChipScope Pro: 15543 cycles!). Moreover opening and closing the connection consumes additional time. The actual data rate can not be determined exactly, because the required time for opening and closing the connection as well as the gap between reading and writing strongly depend on the computer, but it is much less than 2Mbyte/s.

1.2.4 FPGA Configuration and Programming

Xilinx ISE

The Xilinx ISE (Integrated Software Environment) is a CAD (computer aided design) tool for assisting in FPGA design [15]. The ISE includes schematic capture, simulation, implementation and programming tools, which can all be started from the 'project navigator'. The navigator shows all source files, all CAD tools that can be used with the source files, and any output or status messages and files that result from running a given tool (see Fig. 1.9). The ISE works with the simulation tool Modelsim. For VHDL compilation and synthesis the XST tool (Xilinx Synthesis Technology) is used [16]. Pins are assigned using the PACE software (Pinout and Area Constraints Editor). For translating, mapping, placing and routing the Xilinx implementation tools are offered. Finally the implemented design is translated to a bit stream which is programmed to the FPGA. Figure 1.9 also shows the design procedure. Two special development tools, the Xilinx CoreGenerator and the ChipScope Pro

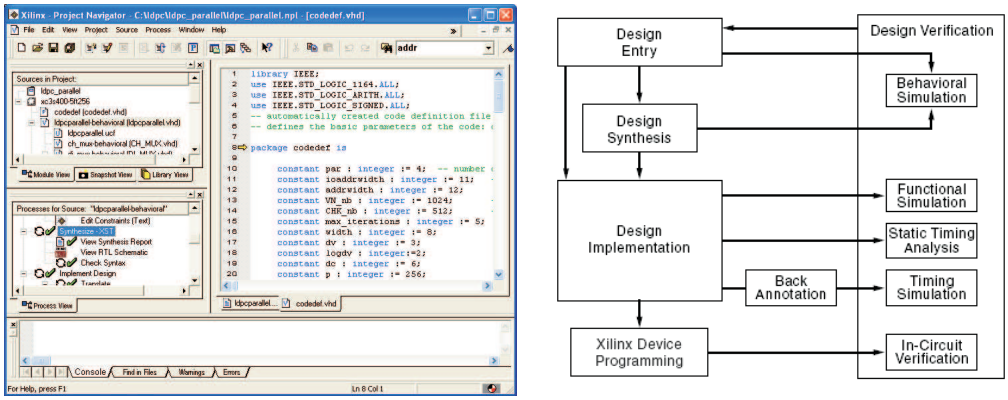

Figure 1.9: ISE and design procedure.

software are described more deeply in the next two sections, as these two tools were intensively used during design procedure.

Xilinx CoreGenerator

The Xilinx CoreGenerator is a design tool that delivers parameterizable cores optimized for Xilinx FPGAs. I used this tool to configure the block RAMs in the decoder. An input file with all required parameters can be offered the CoreGenerator in batch mode. Then, the core with the specified parameters is generated. I used the CoreGenerator in batch mode in order to generate the required input file in Matlab and hand it over to the CoreGenerator.

ChipScope Pro

The ChipScope Pro [17] tools integrate key logic analyzer hardware components with the target design inside the Xilinx Spartan-3 device. The ChipScope Pro tools communicate with these components and provide the designer with a complete logic analyzer. ChipScope consists of the ChipScope Core Generator, ChipScope Core Inserter and the ChipScope Analyzer. The Core Generator provides templates and netlists for the various selectable cores, such as the integrated logic analyzer. The Inserter automatically inserts these templates in a synthesized user design and the Analyzer provides device configuration, trigger setup and trace display for the various cores. The ChipScope Analyzer communicates with the implemented cores via a JTAG interface.

In order to analyze various signals on the FPGA design, the application first has to be synthesized. Next various cores such as an integrated logic analyzer core can be inserted in the synthesized design, and dedicated signals can be selected to be monitored. After implementing and programming the design on the FPGA, these signals can be triggered and monitored with the ChipScope Analyzer. I intensively used this tool since it allows monitoring of various signals on the implemented design, running on the FPGA. Throughout the text ChipScope figures are used to support the comprehension of the realization.

1.3 Architectures

In this section different approaches for practical implementations of LDPC decoders are introduced. First I will present a completely parallel architecture, next, in contrast to that, a serial architecture, and finally a partly parallel one, which compromises the properties of both. The latter two have been realized for my diploma thesis and are described in detail in the Chapters 2 and 3.

1.3.1 Direct Instantiation in Hardware

A very straightforward way to implement an LDPC decoder on hardware is to directly instantiate the SMA on the chip. Then, every single variable node and every single check node is realized as a single processing unit. The interconnection network between variable nodes and check nodes (also called interleaver) is a set of buses connecting the variable nodes with the check nodes according to the parity check matrix H, or the corresponding factor graph. This approach has the advantage that all the variable nodes can process their operations in parallel, as well as all the check nodes. Hence, quite high data throughput can be achieved. Such an LDPC decoder has been implemented on an ASIC in [5], with block length of 1024, reaching a data throughput of 1 Gbit/s and consuming 1.7M system gates. This example already shows the massive resource consumption, which is the big disadvantage of such an implementation. The amount of variable nodes, check nodes and wires which connect the variable nodes and the check nodes are proportional to the block length. Moreover routing may become an unsolvable problem. As the block length is growing, variable nodes and check nodes may be distributed over a large area, and very long wires are needed, which all would have to be placed on the chip. These problems make this approach unattractive for practical solutions with block lengths of 1000 to 10000.

1.3.2 Serial Architecture

A completely different approach is a serial architecture. A block diagram is shown in Fig. 1.10. Channel values are received at the I/O interface and written to an I/O memory which is located in the variable node processor. As soon as one block is received, the variable node processor starts computing the messages. The variable node processor sequentially works on all the variable nodes. The results of each variable node are written to the message memory block, denoted by the RAM block in Fig. 1.10, to consecutive addresses. As soon as variable node processing is completed, the control unit enables the check node processor, which also sequentially works on all check nodes. For each single check node, the incoming messages are fetched from the message memory, processed, and written back to the same locations. When check node processing is finished, the control unit enables the variable node processor again and so on. This continues until all iterations are completed. The check node processor accesses the message memory through the interleaver, which is an address table for indirect addressing. According to the parity check matrix the corresponding address for check node 1, 2, ... are stored. As soon as all iterations are finished, the next block is processed.

This approach has the advantage that it is simple and only consumes quite little resources. On the other hand the throughput is limited, depending on the maximum number of iterations. A compromise of the two described approaches is a partly parallel architecture.

1.3.3 Partly Parallel Architecture

An approach to increase throughput while keeping resource consumption low is the partly parallel architecture. In this architecture several sets of variable node processors, check node processors, interleavers and memories are instantiated, see Fig. 1.11 for a simplified block diagram. These units I call decoding units. Each decoding unit works on a different set of variable nodes and check nodes. If there are two decoding units for a $(3, 6)$, length 1024 LDPC code for example, the first one works on variable nodes 1 to 512 and on the check nodes 1 to 256 while the second decoding unit works on variable nodes 513 to 1024 and on check nodes 257 to 512. Each check node processor has to access both message memories, and as they are working concurrently, access conflicts must be expected. Moreover, due to the randomness in the parity check matrix, a different amount of values from each message memory is needed for each single check node. However, if the code is regularly constructed, the accesses to the memory can be scheduled in order to avoid access conflicts. Different approaches to achieve this are described in Chapter 3.

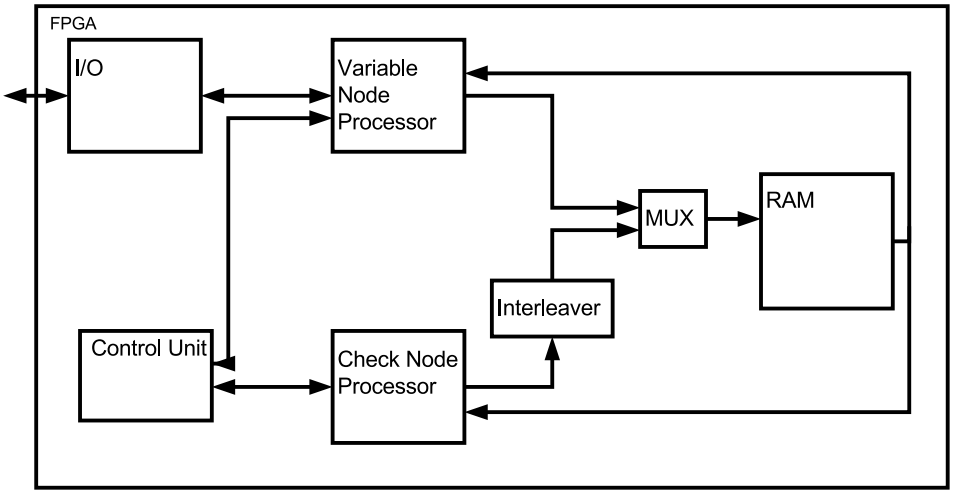

Figure 1.10: Block Diagram of a Serial Architecture.

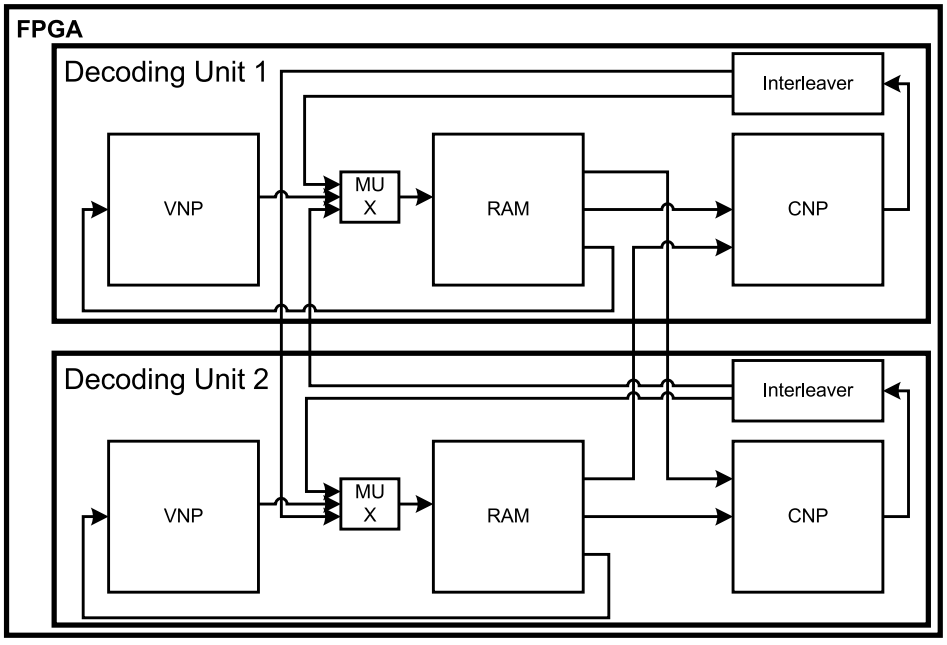

Figure 1.11: Block Diagram of a partly parallel Architecture.

For practical implementations this approach is usually the most attractive, since it compromises resource requirements and data throughput. This trade off offers a nice additional design parameter. Chapter 3 describes my realization of a partly parallel architecture and my code construction mechanism.

2 Serial Architecture

In this chapter a practical realization of a serial architecture for an LDPC decoder is presented. First the different modules of the architecture are described. Next the main problems, regarding memory consumption when using randomly constructed codes and memory access conflicts when partly parallelizing the architecture, are shown. A section about automatic VHDL module construction and component instantiation concludes this chapter.

A block diagram of the serial architecture was shown in Fig. 1.10. For decoding, the Sum - Min algorithm is used, postprocessing, as explained in Section 1.1.1, can be applied. Recall that the variable node processor sequentially works on all variable nodes and writes the results to consecutive addresses in the message RAM. This means, in a (3,6) code, that the variable node processor first processes VN 1 and writes the results to the memory locations 0, 1 and 2. Then the VN 2 is processed, the results are written to locations 3, 4 and 5, and so on. As soon as all variable nodes are finished, the check node processor works sequentially on all check nodes. Therefore, messages are fetched from the message RAM, which is addressed through the interleaver, processed and written back to the same locations. This continues until all iterations are successfully finished. The iteration process could also be interrupted as soon as a valid code word is detected, but this would result in additional logic for checking the results after every iteration. For a system with a fixed data rate this would not draw any advantage, since the system has to be designed for the worst case, and this is the maximum number of iterations for every block. On the other hand, input and output buffers could be used in order to average the data throughput. The output buffer then streams data with constant bit rate, and if a buffer underflow is expected it signals to the decoder that less iterations should be applied. Due to the buffers this yields a larger total delay and could increase the bit error rate, if less iterations are used.

All units and processes of the design are clocked in order to keep synchronization and avoid timing conflicts. The LLRs which are passed between variable nodes and check nodes are represented by the two's complement in the realization. All parameters such as the code parameters, the message width and the number of iterations and some basic functions are defined in a VHDL package

and can easily be changed in order to define a different decoder and get a scalable design. An exact block diagram is shown in Fig. A.1.

2.1 I/O Interface

The 'on chip' I/O interface realization (see Fig. 1.8) is a modified version of the reference design provided by Digilent [6, 7]. The interface provides an 8 bit bi-directional parallel data bus and some handshaking lines to control the data transfer, namely *clock, address strobe, data strobe, write and wait*. Communication is always initiated by the PC in order to keep synchronization. The *write* signal indicates the direction of data transfer. The end of a transfer is indicated by *wait*. *Wait* is driven high on the FPGA side, as soon as a transfer starts and driven low when the transfer is completed. The application running on the PC waits until *wait* is low, then it initiates the next transfer. The logic implements a set of registers, one that is read from, one that is written to, and one address register. A ChipScope sample of the interface is shown in Fig. A.2.

As soon as a transfer is detected, the I/O interface module signals to the variable node processor module (signal channel_in_active) and hands over the data in case of a write cycle (see Fig. A.3). The variable node processor then writes the value to the I/O memory, and increases the address counter. During a block is read, the I/O module again signals a channel use to the variable node processor (signal channel_out_active), which hands over the value which should be read next (see Fig. A.4).

As the width of the internal messages can be chosen, whereas the I/O module only transfers data byte-wise, the widths are mapped in the I/O module. For a selected width of w bits ($w < 8$) the lower w bits are handed over to the I/O RAM, whereas a message with w bits width is extended to 8 bits according to the two's complement, when data is read. Internal widths larger than 8 bit are usually not necessary and therefore not supported.

2.2 Processing Units

The variable and check node processing units, also called variable node processor and check node processor respectively, compute the operations of the Sum - Min algorithm (see Section 1.1.1).

2.2.1 Variable Node Processor

The variable node processor realizes two functions: On the one hand the channel messages are stored, on the other hand it processes the variable node operation.

I/O memory

The I/O memory is instantiated in a block RAM with width equal to the message width. It realizes a swinging buffer (see Fig. 2.1), therefore the addressable size is two times the block length N. The I/O memory can then be considered as two memory blocks (memory 1 and memory 2) of depth N. One memory block is saving the data that is currently worked on (process), the other memory is used for reading the results and writing a new block (read/write). The memory is configured as a dual port memory, therefore it can be simultaneously accessed by the I/O module on the one hand (read/write) and by the processing part on the other hand (process). A one bit signal (buf) is used to store which block is the current I/O block and which one currently is processed. As soon as the transfer of one block to memory 1 is completed, memory 2 is used for reading and writing. First the results are read from memory 2, then new data is written to it. Simultaneously the block in memory 1 is processed. As soon as the data transfer to memory 2 is completed, the blocks are switched again. During the read/write process, the entire block is first read, then the new data block is saved. Channel accesses are always signaled by the I/O module.

It can easily be seen that one block always has to be processed faster than reading and writing (see Fig. 2.1). This can be achieved by selecting a proper number of iterations. This mechanism leads to a fixed total delay of two blocks, one for reading and writing, one for processing. Thus, using longer block lengths leads to longer delay.

Variable node operation

The variable node processor also processes the variable node function (Fig. 2.2, Equ. (1.1)). A ChipScope Pro figure of variable and check node processing can be found in Fig. A.6. In the first cycle the variable node processor fetches its first message from the message memory, buffers it and a *sum* variable is initialized with it. In cycle two the channel value and the second extrinsic value are read and added to *sum*, moreover the extrinsic value is latched. The remaining $d_v - 2$ extrinsic messages are read, added and buffered during the next $d_v - 2$ cycles. The reason why the channel message is read in cycle 2 is

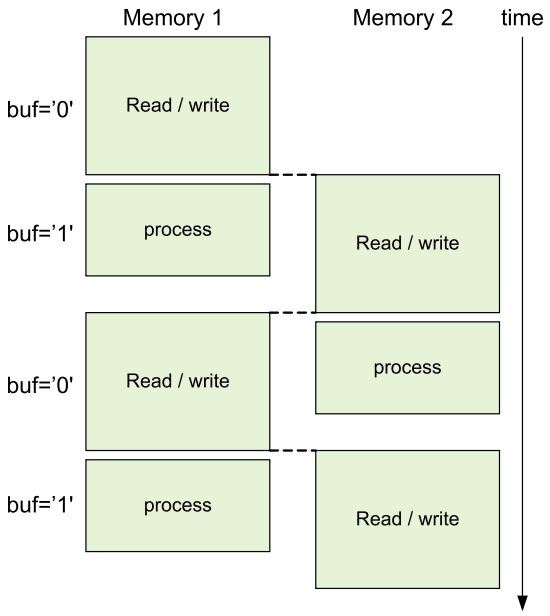

Figure 2.1: I/O memory, swinging buffer.

explained in Section 2.2.4. At this point of time, the *sum* variable holds the sum of all extrinsic values plus the channel value.

During the following d_v cycles the outgoing messages are written back to the message memory. An outgoing message is calculated by taking the total sum and subtracting the corresponding extrinsic value, which is taken from the buffer. The messages are written to consecutive addresses in the message RAM. Hence, for the first variable node, messages are fetched from (and written to) addresses 0 to (d_v-1), for the second variable node from addresses d_v to $(2d_v-1)$, ..., for the last variable node from address $((N-1)\cdot d_v)$ to $(N\cdot d_v-1)$.

As an outgoing message is the sum of all the other incoming messages plus the channel value, the result could be larger than each of the incoming values, and thus an overflow must be expected. In order to avoid this I spent $[\log_2(d_v)]_+$ additional bits for the *sum* variable, where $[\cdot]_+$ denotes a *round up to the next integer* operation. As the width of the outgoing messages than again is smaller, a saturate function is applied to the sum when it is assigned to the outgoing message. The return value y of this function is:

$$y = \begin{cases} 2^{\text{width}-1}-1 & \text{if sum } > 2^{\text{width}-1}-1 \\ \text{sum} & \text{if } -2^{\text{width}-1} < \text{sum} < 2^{\text{width}-1}-1 \\ -2^{\text{width}-1} & \text{if sum } < -2^{\text{width}-1} \end{cases},$$

with sum denoting the input with broader width.

The counter (VN_cntr) which holds the number of the variable node that is currently processed is realized in the variable node processor and is increased as soon as a variable node is finished. When processing of the last VN is completed, the variable node processor signals to the control unit that processing is completed (Fig. A.6, signal VN/CN processing done). Then the control unit increases its iteration counter, deactivates the variable node processor and enables the check node processor (signal VN/CN counter active in Fig. A.6).

During the first iteration no extrinsic information is available. Thus the message memory is deactivated by the control unit during reading and no extrinsic information is added. During the write cycles of the first iteration, the messages from the previous block are replaced by the new messages in the memory block. As it can be seen in Fig. A.6 (enable RAM signal), there is no output of the message memory (DO[0] to DO[7]) in the first iteration during variable node processing.

During the last iteration (which is signaled by the control unit, see Fig. A.6), the results are written back to the I/O RAM. This results are just the sum of all extrinsic messages and the channel message, corresponding to one variable node. When processing is finished, the control unit deactivates the processing units and waits until the transmission of the following block is successfully completed.

2.2.2 Check Node Processor

The check node processor realizes the check node function of the Sum - Min algorithm (Equ. (1.3), Fig. 2.2) for the check nodes. Again, first the messages are fetched from the memory and buffered in the first d_c cycles. For the processing, the absolute value of two minima and the product of the signs of all incoming values are buffered. While those messages are read from the RAM, their magnitudes are compared with the current minima, that are buffered. Thereby, the two smallest absolute values are found. The product of all signs of the incoming values is just an XOR operation of all those signs. The outgoing messages are computed while writing them back to the memory. If the absolute value of the buffered message is the smallest minimum, the other minimum is written to the memory, else the smallest one. The sign is processed by computing an XOR operation of the buffered sign and the sign of the buffered value. In the case of postprocessing, the postprocessing function is applied to the outgoing values. For more details refer to Fig. 2.2.

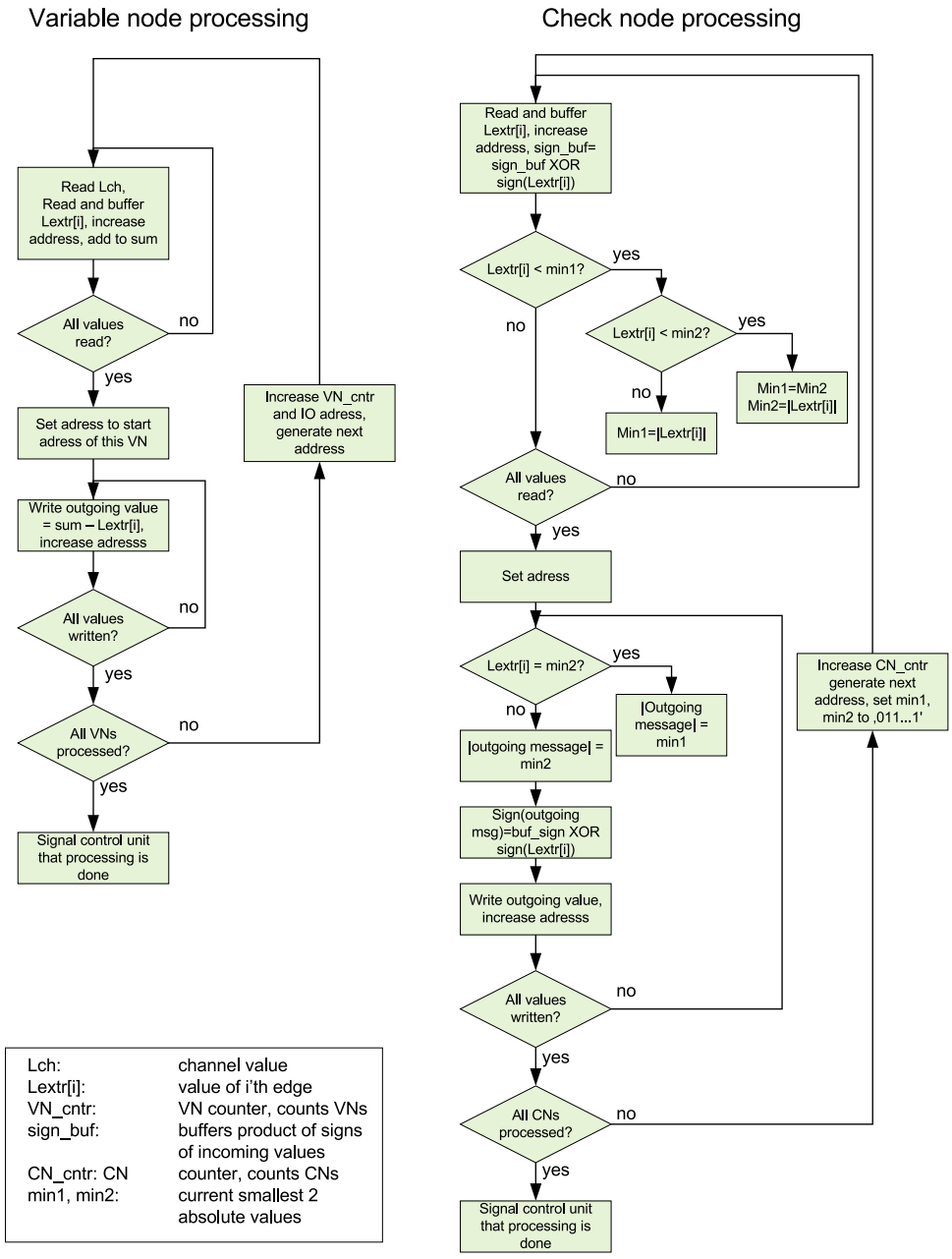

Figure 2.2: Processing of variable and check nodes.

The memory is accessed through the interleaver module, which is an indirect address table and which generates the addresses according to the factor graph. The check node processor simply generates consecutive addresses (equal to the variable node processor), which then are mapped to the needed ones (see Section 2.3).

The check node counter (CN_cntr) is again located in the check node processor and is increased as soon as one check node is finished. When the last check node is finished, the check node counter signals to the control unit (see Fig. A.6, signal VN/CN_processing_done), which then deactivates the check node processor and enables the variable node processor.

2.2.3 Postprocessing

As shown in Chapter 1, the performance loss of the Sum - Min algorithm compared to the Sum - Product algorithm can be undone by applying a postprocessing function to the outgoing check node messages. The exact postprocessing function has to be approximated by a scaling or an offset function, since it is nonlinear and therefore difficult to implement on hardware.

In my design I instantiated both of the postprocessing functions in a VHDL package, and the user has the choice to select one of both or none. Moreover the correction parameter (β for the offset function, α for the linear scaling) can be assigned in the package. For the scaling function only certain values of $1/\alpha$ can be assigned due to finite data precision, for example 0,75 or 0,875. The scaling function was only verified using Modelsim and ChipScope Pro, but not by comparing the results with the results of a Matlab script, that also does the same decoding, since Matlab works with floating point numbers. The offset functions work well.

2.2.4 Pipelining

Since all memories can be configured as dual port memories, it is possible to access the message memory twice in a cycle. Hence, the variable and check node processor can already fetch the values for VN(n+1), and CN(n+1) while the results for VN(n) and CN(n) are written. This doubles the throughput of the design. On the other hand some of the internal buses and registers (like sum, Lextr, min1, min2, ..., see Fig. 2.2) of the variable node processor and check node processor have to be duplicated, because two VNs / CNs are always processed concurrently. The message RAM is configured on port A for reading only, and on port B for writing only.

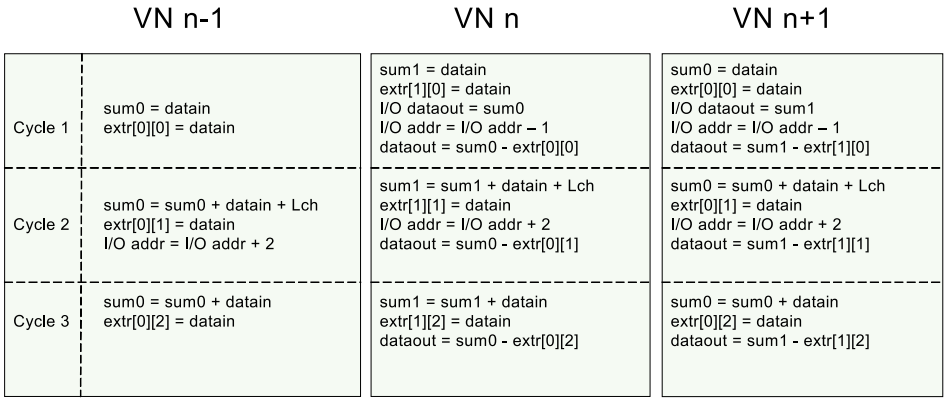

Figure 2.3: Pipelined processing of variable nodes (d_v=3).

Figure A.8 shows a ChipScope sample of the pipelined variable node processing for a $(3, 6)$ length 32 LDPC code. During the read cycles of VN 1 there is no data to be written, this is the time when the pipeline has to be filled, and which causes additional processing time. As soon as the pipeline is filled, processing lasts d_v cycles per VN, which are needed to fetch the d_v values from the message RAM. The channel message is stored in the I/O memory, therefore it can be read simultaneously with the extrinsic values.

Note that the latency for a memory access is two cycles, because all modules are clocked. This means, when an address is generated in the variable or check node processor at clock cycle n, it is passed to the bus connecting the processing unit with the RAM at cycle $n + 1$, and the result can be read at cycle $n + 2$! During the last iteration the results (total sum of extrinsic messages plus channel message) need to be written back to the I/O memory. This happens simultaneously with writing back the first extrinsic value. This is the reason why the channel message is always read and added during the second cycle. The I/O address is applied at the last cycle of the previous variable node processing time, i.e. at cycle d_v. Then the valid result can be read at cycle 2 of the next processed variable node. If it was read at cycle 1, the address would have to be applied at cycle $d_v - 1$, but if a $(2, 4)$ code was used, this would be the first cycle of processing the previous variable node. Note that in cycle 1 already the result is written to the channel memory, and therefore another address has to be written. Hence, access conflicts would occur. Note that this problem does never occur if a higher variable node degree is used. All in all d_v cycles are needed for processing one variable node, as soon as the pipeline is filled, also see Fig. 2.3

Figure A.9 shows a ChipScope figure of the pipelined check node processing, again for a (3,6) length 32 code. Until the signal write enable (WE port B) goes high, the pipeline has to be filled. As soon as the pipeline is filled, messages are read and written to the memory in parallel. Thus, processing time of one CN is exactly d_c cycles.

The additional processing time caused by filling the pipeline is $d_v + 2$ cycles for variable node processing and $d_c + 3$ cycles for check node processing. As already mentioned, the latency from generating an address internally and reading a valid result is 2 cycles. Due to the interleaver an additional cycle is required during check node processing.

2.3 Memories

In this design I use block memories for buffering the channel values (I/O memory), the messages that are passed between variable and check nodes (message memory) and for realizing the interleaver. All memories are configured as dual port memories, in order to support a pipelined design.

2.3.1 Message and I/O Memory

The amount of messages that has to be stored is $N \cdot d_v$, which is the needed depth of the message memory. As already mentioned, the I/O memory is of depth $2N$, since one block is always processed while the other one is transfered. The width of the two memories is arbitrary (up to eight bits), and can be set in a VHDL package, where all the code and design parameters are listed. A width larger than eight is usually not needed, and therefore not implemented. Note however, that the block RAMs can only be configured with 1, 2, 4 or 8 bits width on Xilinx FPGAs. Thus, a chosen width of 6 bits also requires a block RAM with 8 bits width, and therefore no memory space is saved. On the other hand less logic is needed for registers, buses, adders, ... of course.

I configured the memories using the CoreGenerator in batch mode, handing over an input file which is automatically generated by Matlab (also see Section 2.5). The CoreGenerator automatically cascades several memories in order to achieve the required depths and widths.

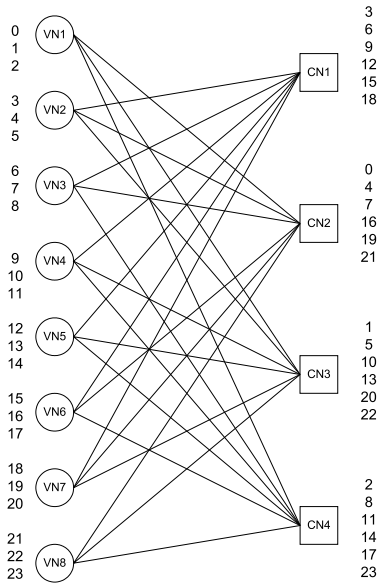

Figure 2.4: Interleaver.

2.3.2 Interleaver

The interleaver is an address table which stores the addresses of the check node edges according to the parity check matrix. It is realized as a ROM. This is demonstrated by Fig. 2.4 for a (3,6) length 8 code. The variable nodes read and write their messages to consecutive addresses, as shown on the left hand side. The check nodes need to access the memory at the addresses which correspond to the factor graph, as shown on the right hand side. The check node processor achieves this by generating consecutive addresses for each check node edge, similar to the variable nodes, i.e. $0, 1, \ldots, 5$ for CN 1, $6, 7, \ldots, 11$ for CN 2, $\ldots 19, 20, \ldots, 23$ for CN 4. The interleaver ROM is then addressed with these addresses and returns the needed addresses, i.e. $3, 6, 9, 15, 18$ for CN 1, $0, 4, 7, 16, 19, 21$ for CN 2, and so on.

I realized the interleaver as a block RAM, which is configured as a ROM. The address table is generated in Matlab. For a pipelined design the interleaver moreover needs to be configured as a dual port ROM.

Note the amount of memory needed by the interleaver compared to the memory consumed by I/O and message RAM: The number of messages that has to be stored is $N \cdot d_v$ for a regular code, the amount of channel messages is $2 \cdot N$. For both, the width can be selected arbitrarily. This means, that the memory consumed by this two modules is proportional to the block length, it is $N \cdot (d_v + 2) \cdot width$. On the other hand $A = [\log_2(N \cdot d_v)]_+$ bits are

needed to address the data of the message memory, therefore the width of the interleaver ROM has to be A. The interleaver depth is equal to the message memory depth, $N \cdot d_v$. Thus, the total amount of memory consumed by the interleaver is $[\log_2(N \cdot d_v)]_+ \cdot N \cdot d_v$, which is growing *superproportionally* in the blocklength.

This should be clarified by an example: a block length N of 4096 and a variable node degree d_v of 3 lead to $A = 14$ bits for addressing the 12288 memory locations in the message memory. Then, if a width of e.g. 8 bits is chosen, the memory consumed by the message memory is $8 \cdot 12288$ bits, whereas the amount of memory consumed by the interleaver is $14 \cdot 12288$ bits, which is almost twice the amount consumed by the message memory!

If somehow deterministically constructed parity check matrices are used, the interleaver may be derived from a set of parameters and the entire parity matrix (which corresponds to the interleaver) does not have to be saved. This approach becomes very attractive for larger block lengths, and is discussed in Chapter 3.

2.4 Memory Access in Parallel Architectures

Throughput could be further increased by using several variable node processors and several check node processors, which leads to a parallel architecture. But then all variable node processors and all check node processors work on the same message set, stored in the message memory. The memory can only be accessed once per cycle (or 2 times if configured dual ported and pipelined). Hence, only one unit could work on the messages, and not all concurrently. This shows, that the message memory is the bottleneck of the architecture.

However, each variable node unit and each check node unit could be equipped with its own interleaver and block RAM for storing and accessing the messages. Such units, consisting of a variable and check node processor, an interleaver and a message RAM, I call a decoding unit. Then all variable node processors and all check node processors of each unit could access the memories concurrently. In a $(3, 6)$ length 16 code with two parallel units, the variable node processor of unit 1 works on variable nodes 1 to 8, while the second variable node processor works on variable nodes 9 to 16. The check node processor of unit 1 works on check nodes 1 to 4, the second check node processor on check nodes 5 to 8. This architecture is shown in Fig. 1.11.

The variable node processors read and write their messages to the message memories included in the decoding unit, whereas the check node units need to access *all* memories! Thus, it is no problem to process the variable nodes concurrently, whereas for processing the check nodes in parallel, access conflicts have to be expected. If however, a code is deterministically constructed, this problem can be taken into account. All check nodes can then access different memories concurrently. This issue is addressed in Chapter 3.

2.5 Code Generation and Module Instantiation

This section explains the automatic code generation and VHDL module configuration process, and the block RAM instantiation with the CoreGenerator in batch mode. The I/O module, the variable and check node processor and the control unit are equal for all different codes, except to some parameters such as width, number of variable and check nodes, variable and check node degree d_v, d_c maximum number of iterations, and the addresswidths. As all of these parameters are stored in a VHDL package (named codedef.vhdl), the other modules need not to be changed. Thus, only the VHDL package and the memory modules need to be configured correctly.

The code generation process now works by doing the following steps:

- First a *random* code with the desired parameters is created: this is done with the Matlab DLL (mex function) *c = createsparsepchk(lambda, rho, N, [triangular, no4cycles])*, written by Gottfried Lechner. N denotes the block length, lambda and rho the variable and check node degree distribution respectively, and triangular and no4cycles are two options to triangular the parity check matrix H and to avoid cycles of length 4. Another representation of the code can be derived by calling *ldpc=sparse2ldpccode(c)*, where ldpc is a Matlab structure array (struct), containing the variable and check node degrees, and the interleaver.

- The Matlab script *createinitfile(ldpc.interleaver)* generates file init.coe with the ROM coefficients for the interleaver ROM. This file is needed by the Xilinx CoreGenerator.

- The three routines *createinterleaver(length, dv)*, *createmsgRAM(length, dv, width)* and *createioRAM(length, width)* generate the required CoreGenerator input files for instantiating the interleaver ROM, the message

and the I/O RAM respectively. These files hold the required parameters such as width and depth of the memory, single or dual port configuration, RAM or ROM usage, the FPGA device, used pins and their polarities, the selected primitive and so on.

- Next the CoreGenerator is started in batch mode with the command *dos('coregen -b projectpath\ioram.txt -p projectpath')*, which instantiates all the memories.
- Finally the VHDL package codedef.vhdl is written (see Appendix A.1) by calling the Matlab script *createpackage(N, width, dv, dc, it, mode, corr)*. N again denotes the code length, width the desired message width, dv and dc the variable and check node degree, it the maximum number of iterations. Mode can be selected to be 0, 1 or 2, where 0 configures the postprocessing function to the linear scaling, 1 is the offset function and 2 is no postprocessing used. Finally, corr denotes the correction term for postprocessing (only working for the offset function, the linear scaling is set to 0,875).

In addition to the parameters defined in codedef.vhdl, a set of functions that is used in the design is defined in codedef.vhdl. *Saturate(x)* realizes the saturation function needed for adding in the variable nodes (described in Section 2.2.1), *log2(x)* returns the width needed to map an unsigned integer to a standard logic vector, and *postproc(x)* is the selected postprocessing function (either scaling, offset or none). Moreover there are two array type definitions which are needed to buffer the incoming messages during variable and check node processing.

All these routines can be called from the single script *[ldpc, c, t] = createldpc(length, dv, dc, width, max, it, mode, corr)*. Ldpc and c again define the code whereas t is a randomly generated testblock that can be used for testing the design. Hence, only the code and architectural parameters need to be given to Matlab in order to get the required VHDL package that defines the entire decoder and to instantiate the needed components. Then the design only has to be synthesized and programmed to the FPGA in order to get a working decoder.

2.6 Results

Throughput

For processing one block, the following processing time is consumed: *(variable node processing time + check node processing time)*(number of itera-*

tions)+(result output).

The variable node processing time is $d_v \cdot N$ cycles for processing N variable nodes when the pipeline is filled, and $d_v + 2$ cycles for filling the pipeline. Check node processing requires $d_c \cdot M$ cycles for processing the check nodes, and additional $d_c + 3$ cycles for filling the pipeline. Result output requires the same time as variable node processing. This leads to the total processing time T_b of one block:

$$\begin{aligned} T_b &= \left(\underbrace{(d_v \cdot N) + (d_v + 2))}_{\text{VN processing}} + \underbrace{(d_c \cdot M + (d_c + 3))}_{\text{CN processing}} \right) \cdot It \\ &\quad + \left(\underbrace{(d_v \cdot N) + (d_v + 2)}_{\text{result output}} \right), \end{aligned} \tag{2.1}$$

with It denoting the number of iterations.

For a $(3, 6)$ length 4096 code, using 10 iterations for processing, this leads to

$$\begin{aligned} T_b &= ((3 \cdot 4096 + 5) + (6 \cdot 2048 + 9)) \cdot 10 + (3 \cdot 4096 + 5) \\ &= 258193 \text{ cycles/block.} \end{aligned}$$

At a clock frequency of 50MHz this leads to $4096 \cdot 50 \cdot 10^6 / 258193 = 793,2$Kbit/s. For a filled pipeline and without regarding the last iteration $d_v + d_c \cdot (d_v/d_c) = 2d_v = 6$ cycles are required per iteration.

By using parallel architectures, the throughput can be further increased, depending on the number of parallel units (see Chapter 3).

Resource Consumption

The consumed resources of a $(3, 6)$ length 1024 and a $(3, 6)$ length 2048 LDPC code, using an internal message width of 8 bits, are shown in Table 2.1. The two examples show, that for growing blocklengths logic consumption more or less is constant, whereas memory requirements are increasing excessively. The delays at the critical paths are 5,56ns and 7,83ns for the length 1024 and length 2048 code, allowing maximum frequencies of 180MHz and 127,7MHz respectively!

In order to allow a comparison with quite often used, larger FPGAs, such as the Xilinx Virtex II Pro family, I synthesized the two example codes on a

Table 2.1: Resource consumption of $(3, 6)$ length 1024 and 2048 LDPC codes.

	total available	Code $N = 1024$	Code $N = 2048$
Slices	3584	471 (13%)	491 (13%)
Slice Flip Flops	7168	368 (5%)	380 (5%)
4 input LUTs	7168	843 (11%)	890 (12%)
block RAMs	16	6 (37%)	10 (62%)

2vp30fg676-5 device (Virtex II Pro). The consumed resources can be found in Table 2.2. Note, that similar to the Spartan 3 FPGA also on the Virtex II Pro device the block RAMs are the constraining elements of the implementation.

Table 2.2: Resource consumption of $(3, 6)$ length 1024 and 2048 LDPC codes, synthesized on a Virtex II Pro FPGA.

	total available	Code $N = 1024$	Code $N = 2048$
Slices	13696	447 (3%)	479 (3%)
Slice Flip Flops	27392	365 (1%)	379 (1%)
4 input LUTs	27392	806 (2%)	868 (3%)
block RAMs	136	6 (4%)	10 (7%)

3 Parallel Architecture

3.1 Code Construction

As already mentioned a deterministically constructed code has very important advantages regarding memory access in partly parallel architectures and derivation of the interleaver from a set of parameters. This should be achieved without any loss of decoding performance compared to randomly constructed codes. There are two major influences that limit the decoder performance: First, the minimum distance of the code has to be sufficiently large, which is usually satisfied if the block length is large enough (in the range of practical applications, 1000 to 10000). Secondly, the Sum - Product and Sum - Min algorithm only work perfectly if the parity check matrix is cycle free, which can never be satisfied. But if the cycles in the graph are large enough, the performance of the algorithms is still good. Therefore the size of the smallest cycle in the parity check matrix (also called girth), plays an important role.

In this section I first introduce an approach by H.Zhong [14], and then present the construction method that I used for realizing a partly parallel decoder.

3.1.1 Approach 1

H.Zhong and T.Zhang [14] developed a code construction method for solving the two problems of mapping the code structure to a partly parallel architecture and deriving a partly deterministic interleaver. First, a base parity check matrix H_b of size $M_b \times N_b$ is constructed, which is then randomly expanded by using cyclic shift matrices. Every '1' in H_b is replaced by a cyclic shift matrix with a random cyclic shift, and every '0' with an zero matrix. This is shown in an example for a $(2, 4)$ code with length 32, using a base matrix of size 4×8:

$$H_b = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

Now every '1' in H_b is replaced by a cyclic shift matrix with a random cyclic shift T of size $p \times p$, e.g. 4×4:

$$T = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix},$$

with

$$T^0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, T^1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \ldots, T^3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

where the exponent is a random integer in [0, p-1], e.g:

$T_{12} = T^1$, $T_{15} = T^1$, $T_{16} = T^0$, $T_{17} = T^1$, $T_{21} = T^3$, $T_{22} = T^0$, $T_{23} = T^3$, $T_{26} = T^1$, $T_{31} = T^2$, $T_{33} = T^3$, $T_{34} = T^2$, $T_{38} = T^1$, $T_{44} = T^2$, $T_{45} = T^0$, $T_{47} = T^1$, $T_{48} = T^0$. T_{ij} denotes the cyclic shift matrix replacing the nonzero element in row i and column j of H_b. In addition to that every '0' in H_b is replaced by a zero matrix of the same size $p \times p$. This leads to the following expanded matrix parity check matrix:

$$H = \begin{pmatrix} 0000 & 0100 & 0000 & 0000 & 0100 & 1000 & 0100 & 0000 \\ 0000 & 0010 & 0000 & 0000 & 0010 & 0100 & 0010 & 0000 \\ 0000 & 0001 & 0000 & 0000 & 0001 & 0010 & 0001 & 0000 \\ 0000 & 1000 & 0000 & 0000 & 1000 & 0001 & 1000 & 0000 \\ 0001 & 1000 & 0001 & 0000 & 0000 & 0100 & 0000 & 0000 \\ 1000 & 0100 & 1000 & 0000 & 0000 & 0010 & 0000 & 0000 \\ 0100 & 0010 & 0100 & 0000 & 0000 & 0001 & 0000 & 0000 \\ 0010 & 0001 & 0010 & 0000 & 0000 & 1000 & 0000 & 0000 \\ 0010 & 0000 & 0001 & 0010 & 0000 & 0000 & 0000 & 1000 \\ 0001 & 0000 & 1000 & 0001 & 0000 & 0000 & 0000 & 0100 \\ 1000 & 0000 & 0100 & 1000 & 0000 & 0000 & 0000 & 0010 \\ 0100 & 0000 & 0010 & 0100 & 0000 & 0000 & 0000 & 0001 \\ 0000 & 0000 & 0000 & 0010 & 1000 & 0000 & 0100 & 1000 \\ 0000 & 0000 & 0000 & 0001 & 0100 & 0000 & 0010 & 0100 \\ 0000 & 0000 & 0000 & 1000 & 0010 & 0000 & 0001 & 0010 \\ 0000 & 0000 & 0000 & 0100 & 0001 & 0000 & 1000 & 0001 \end{pmatrix}.$$

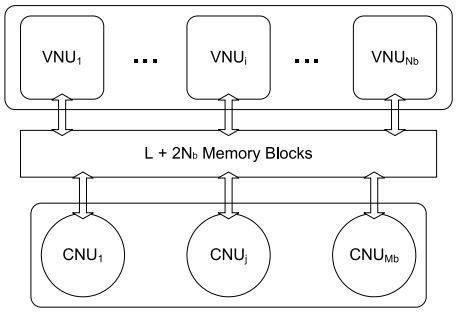

Figure 3.1: Partly Parallel Architecture for constructed code

By expanding H_b with random cyclic shift matrices, the weight distribution is maintained. This means, H_b needs to be a matrix with $d_v = 3$ and $d_c = 6$ in order to create a (3,6) code. Several different matrices H can be generated, since the exponent of the cyclic shift matrices are random. The matrix with the best properties regarding girth can then be chosen.

This matrix can be mapped to the partly parallel architecture shown in Fig. 3.1. It uses N_b variable node processing units and M_b check node processing units, corresponding to the rows and columns in H_b respectively. All variable node units are working concurrently, as well as all check node units. In order to avoid memory access conflicts, each variable and check node unit works on its own memory blocks. This means, that for every intersection of columns corresponding to one variable node processor and rows corresponding to one check node processor in H a separate RAM block is used (except of those that are zero blocks). This is equal to the nonzero positions in H_b. Thus, L different block memories are used for storing the messages, where L is the number of '1s' in H_b. Moreover each variable node unit needs its own I/O memory block, therefore additional $2N_b$ memory blocks are needed.

The smallest size of H_b that can be used for a (3,6) code is 3×6. Then H_b only contains '1s'. Note that this matrix is not sparse, but the expanded matrix H is. This results in $2N_b + L = 12 + 18 = 30$ needed memories! As already mentioned, the Spartan 3 FPGA only exhibits 16 block RAMs. If those are all configured as dual ported memories this could be regarded as 32 block RAMs, but it is impossible to use a larger size of H_b, which reduces the code design parameters. Moreover the memory is not used efficiently, since during variable node processing six units work in parallel on six memory blocks, the other 12 block RAMs are not used, during check node processing three check node units are working in parallel, always accessing three memory blocks simultaneously, while 15 block RAMs are idle. In the next section I will show

a code construction scheme that can be mapped more efficiently to a partly parallel architecture.

By constructing the code as described above, the interleaver can be derived from the base parity check matrix H_b and the exponents of each random shift matrix. Compared to a randomly constructed code, lots of memory can be saved. The exact value depends on the code length and on the size of the base matrix used. For the example given in Section 2.3.2, $14 \cdot 12288$ bits are needed to store the interleaver of the randomly constructed code. If H_b has size 3×6, 18 values of width $[\log_2(18)]_+$ need to be saved in order to define H_b, and 18 values of width $[\log_2(p)]_+$ for the random shifts. In Section 2.3.2 a codelength of 4096 was assumed, this leads to $p = N/N_b = 683$ and $[\log_2(p)]_+ = 10$. Hence, the needed memory to derive the interleaver is $18 \cdot 5 + 10 \cdot 18 = 15 \cdot 18 = 270$ bits, which leads to 0,16% of the memory consumed for storing the interleaver if randomly constructed! Note however, that H_b of size 3×6 is only chosen in this example because it is the only code that can be mapped to the Spartan 3 device. A size of 4×8 is usually more interesting, as then also p is a power of two, if N is a power of two. Then $24 \cdot (9+5) = 336$ bits need to be saved in order to derive the interleaver (0,2% of randomly constructed code in the example). The address generation process is explained in Section 3.2.

3.1.2 Approach 2

By constructing a code as described above, a nice way to derive a deterministic interleaver is found. On the other hand the code can not be mapped to a partly parallel architecture very efficiently. With the following code construction approach, I optimized the constructed code for mapping to a partly parallel architecture. In order to achieve this, I also use the matrix expansion method described above, but do not construct the base matrix randomly. If the base matrix can efficiently be mapped to a partly parallel architecture, it is also possible for the expanded matrix! Hence, I impose certain constraints to the base matrix construction and then continue with expanding the matrix as described above.

Mapping to a partly parallel architecture

By parallelizing the serial architecture as explained in Section 2.4, memory access conflicts must be expected. This should be clarified by an example, of a $(3, 6)$, length 8 LDPC code:

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

By using two parallel units for decoding, unit 1 works on variable nodes 1 to 4 and check nodes 1 and 2 while the second unit works on variable nodes 5 to 8 and check nodes 3 and 4. Figure 3.2 shows the factor graph and the parity check matrix with the corresponding decoding units. Moreover the memory addresses with the corresponding memory in brackets in the factor graph are shown. Since every variable node unit works on its own memory, variable node processing is no problem. But during check node processing both units access both memories. In this example, CN1 accesses RAM 1 two times and RAM 2 four times, while CN3 accesses both RAMs three times. Hence, RAM 1 is accessed 5 times, while RAM 2 is accessed 7 times. During processing of CN2 and CN4 RAM 1 is accessed seven times and RAM 2 five times. Each check node needs to access the memories d_c times, thus we desire that each memory is accessed *exactly* d_c times. In the parity check matrix H this means, that the number of '1s' in the first four columns of row 1 and row 3 must be $d_c = 6$, as well as the first four columns in rows 2 and 4. Then, the number of connections from CN1 to RAM 2 is equal to the number of connections of CN3 to RAM 1. In this case scheduled access of both decoding units to both memories would be possible without any conflicts.

A working example of a $(3, 6)$, length 8 LDPC code with 2 parallel units is shown in Fig. 3.3. While CN1 accesses RAM 1 two times and RAM 2 four times, CN3 accesses RAM 2 four times and RAM 2 two times. Hence, both RAMs are accessed $d_c = 6$ times while processing the two check nodes concurrently. The access of RAM 1 of CN 1 and CN 3 is marked with the rectangle in Fig. 3.3. CN 2 and CN 4 access both RAMs 3 times, which also leads to a total access of 6 times per RAM. Thus, a parallel architecture works on this code. CN 1 can first fetch its messages from RAM 1 (two accesses) while CN 3 first accesses RAM 2 (two times). Next CN 1 works on RAM 2 (four accesses needed) while CN 3 accesses RAM 1 four times. The same can be done for CN 2 and CN 4. This code can be mapped to a partly parallel architecture with 2 parallel units, as shown in Fig. 1.11.

A code can always be mapped to this parallel architecture if the following constraint is fulfilled:

The sum of '1s' in all subrows must be equal to d_c.

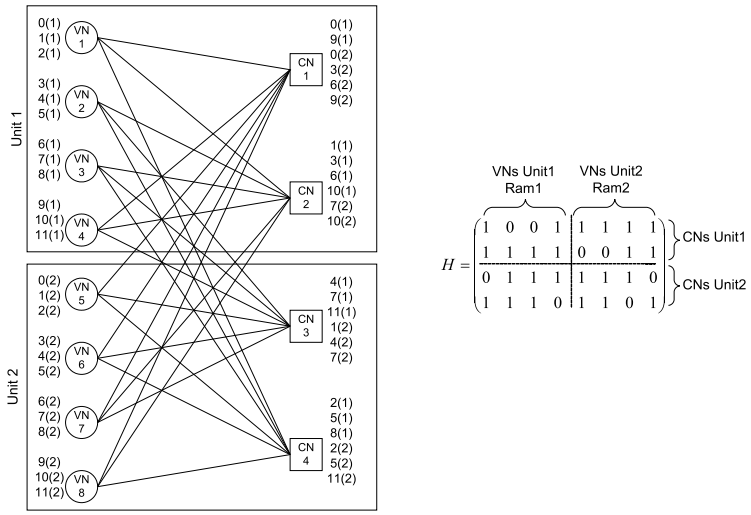

Figure 3.2: Factor graph and parity check matrix for 2 parallel units.

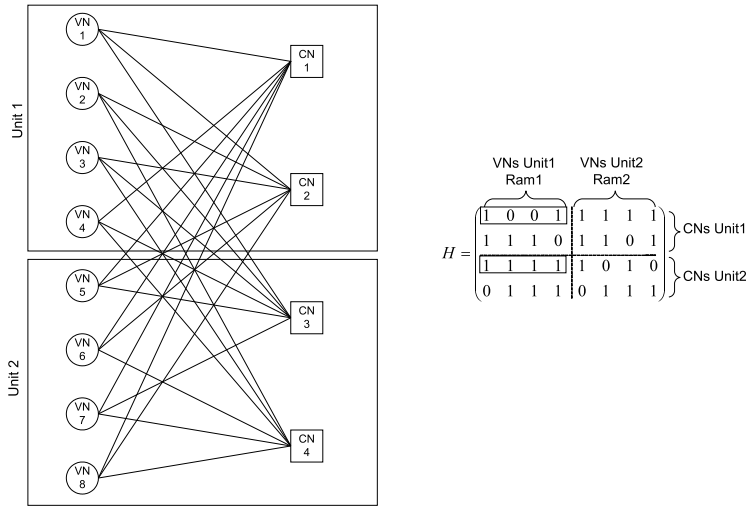

Figure 3.3: Factor graph and constrained parity check matrix for 2 parallel units.

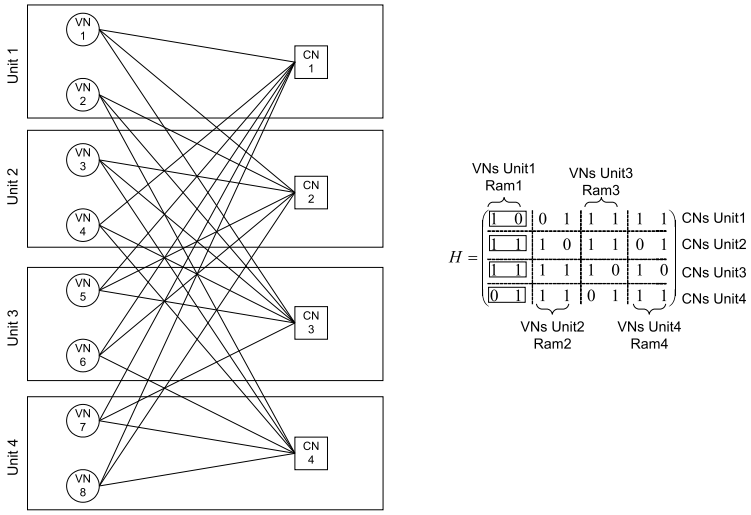

Figure 3.4: Factor graph and constrained parity check matrix for 4 parallel units.

A subrow here is the intersection of all columns corresponding to one RAM with the first, second, $\ldots (M/par)^{th}$ row of all rows corresponding to one decoding unit, as indicated by the rectangles in Fig. 3.3. The rectangles show the intersection of the first CNs of each decoding unit with the block RAM of decoding unit 1. The constraint also has to be fulfilled for the second CNs of each decoding unit accessing RAM 1, which are the two blocks below the marked ones. The same has to be true for the RAM of decoding unit 2 of course. An example of a $(3, 6)$, length 8 code with four parallel units is shown in Fig. 3.4. The rectangles again denote the accesses to RAM 1 during concurrent processing.

Note that the constraint is always fulfilled if a maximum amount of parallelization is used. This means, that each parallel unit only processes one check node. In the example of Fig. 3.4 the 'sum of subrows' is the sum of the first 2 columns, which is always d_c in a $(3, 6)$ code. Not only $(3, 6)$ codes can be mapped to a partly parallel architecture if the above constraint is fulfilled, but all regular codes.

Code Derivation

As already mentioned, a matrix that is created with the matrix expansion method (see Section 3.1.1) can be mapped to the same architecture as the base matrix that it was created of. Therefore I again use this method to

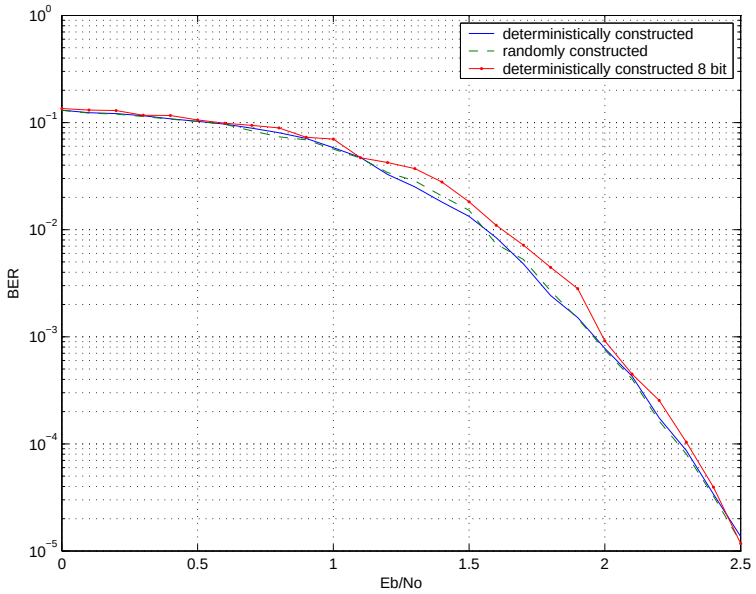

Figure 3.5: decoding performance of a deterministically and a randomly constructed code.

create a matrix of the desired size. Then, also a partly deterministic interleaver can be derived from the base matrix H_b and the applied random shifts.

The difference to the above derived code now is, that the parallel architecture is a different one. Each variable node processor works on one message memory, all check node processors work on all memories concurrently. Thus, memory accesses must be scheduled (see Section 3.3), but it is possible to realize this. In contrast to the above architecture, the required memory blocks now are only one (dual ported) I/O memory per variable node processor and one message memory per decoding unit. This leads to $2 \cdot par$ required block RAMs, with par denoting the number of parallel units.

The decoding performance of this constructed code can be seen in the simulation of Fig. 3.5. A randomly constructed code and a partly deterministically constructed code were simulated. Moreover the decoding performance of the deterministically constructed code, representing the messages with 8 bits and a quantization interval of 0,1, is shown. Note that for a (3,6) length 1024 LDPC code, using a base parity check matrix H_b of size 4×8 the decoding performance is basically the same.

3.2 Interleaving

With the two described code construction methods a way is found to derive the interleaver from the base matrix H_b and a matrix RS saving the random shifts of the cyclic shift matrices. This works for *both* code construction schemes, as they only differ in the way of finding the base matrix H_b, which has to be saved in any case, whereas the expansion method is used in both cases.

The derivation of the interleaver can most easily be seen in an example. Therefore the example of the constructed code in Section 3.1.1 is reconsidered. Recall that each variable node writes the messages to consecutive addresses, e.g. to addresses 0,...,$d_v - 1$ for VN 1, d_v,...,$2d_v - 1$ for VN 2, and to addresses $(N-1)d_v$,...,$Nd_v - 1$. VN m then starts processing its messages at address $(m-1)N$. Hence the interleaver I_b of H_b in the example is:

$$H_b = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}, I_b = \begin{pmatrix} 2 & 8 & 10 & 12 \\ 0 & 3 & 4 & 11 \\ 1 & 5 & 6 & 14 \\ 7 & 9 & 13 & 15 \end{pmatrix},$$

where every row of I_b holds the addresses for the corresponding CN, e.g. CN 1 accesses memory locations 2,8,10 and 12.

By expanding the matrix H_b with cyclic shift matrices of size $p \times p$, each nonzero element in H_b is replaced by a cyclic shift matrix with a random cyclic shift. Those random cyclic shifts are saved in the matrix RS. Then the matrix RS of the example is:

$$RS = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 3 & 0 & 3 & 1 \\ 2 & 3 & 2 & 1 \\ 2 & 0 & 1 & 0 \end{pmatrix}.$$

Every column of H_b is replaced by a column containing matrices of size $p \times p$, which I will call 'block columns' and each row of H_b is replaced by a 'block row'. In block row 1 of the example the first nonzero block is block 2, according to H_b. It holds the values for variable nodes $p+1$ to $2p$ and for check nodes 1 to p. As already mentioned, VN m processes the messages of memory locations $(m-1) \cdot d_v$ to $m \cdot (d_v - 1)$. Hence, the p variable nodes of block column 1 process messages at locations 0 to $p \cdot d_v - 1$, and the p variable nodes of block column x process the messages of locations $p \cdot x \cdot d_v$ to $p \cdot (x+1) \cdot d_v - 1$. The starting address of the xth block column then is $p \cdot x \cdot d_v$. This automatically

is the starting address of the first nonzero block in this block column. For the following $d_v - 1$ nonzero block in this block column the starting addresses then are $p \cdot x \cdot d_v + 1$, ..., for the last nonzero block it is $p \cdot x \cdot d_v + d_v - 1$. Actually this is pretty much the same as in the base interleaver I_b. Therefore the starting address of the block of block row x and block column y can be derived by the base interleaver I_b by

$$\underbrace{(I_b(x,y) - I_b(x,y) \bmod d_v) \cdot p}_{\text{A}} + \underbrace{I_b(x,y) \bmod d_v}_{\text{B}}, \tag{3.1}$$

where term A denotes the starting address of the first nonzero block in a block column, and term B denotes the increase for the starting address of all following nonzero blocks. With this, we calculated the very first address of a block matrix.

This shall be clarified by the example: the nonzero element (3,4) corresponds to element (3,3) in I_b and RS, as it is the third nonzero element in the third row of H_b. Therefore the corresponding address is 6, and the exponent of the random shift matrix is 2. Hence, the '1' is replaced by the cyclic shift matrix:

$$T_{34} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix},$$

The very first address of this block can be calculated with Equ. (3.1), which is $(6 - 6 \bmod 2) \cdot 4 + 6 \bmod 2 = 24$, as $p = 4$ and $d_v = 2$ in this example. This now is the address for the first VN of this block, which can be compared with the expanded matrix. Element (10, 13) of the expanded parity check matrix is the searched one, the corresponding address of this element is in fact 24 (the first nonzero element of column 13 corresponds to the address $(13 - 1) \cdot d_v$).

With the starting addresses of the blocks we already did the most important step. What is still missing, is the derivation of the other addresses, and, what we are looking for, the addresses for the check nodes. The addresses of the other '1s' in the block can simply be calculated by adding d_v for each column, since there are d_v elements in each column. What is still not found, are the addresses for the check nodes, which means the addresses of the '1s', *row-wise* counted. If the starting address of the example block was 24, which corresponded to element (3,1) of T_{34}, then the address of element (1,3) simply is 28, as it is shifted by two columns. Note that two is the corresponding random shift. Hence the starting address for the check nodes is the starting address of the block plus the random shift times d_v.

All addresses of the elements (x, y) of the expanded matrix H are calculated by:

$$I(x,y) = \underbrace{(I_b(x_b,y_b) - I_b(x_b,y_b) \bmod d_v) \cdot p + I_b(x_b,y_b) \bmod d_v}_{\text{Equ.(3.1)}} + \underbrace{+ d_v\,((RS(x_b,y_b) + x \bmod p) \bmod p)}_{\text{address within a block}}, \quad (3.2)$$

where x_b and y_b denote the indizes of H_b and RS that correspond to the blocks of the expanded matrix H.

Since the first two terms of Equ. (3.2) contain a mod d_v operation, it is not easy to implement this directly in hardware. Usually only modulo operations of a power of two can be synthesized. In order to avoid a huge amount of logic for calculating the modulo operation for arbitrary d_v, I precalculated the operations of the first two terms in Matlab and saved their results in a matrix (called *base* in the realization). With this matrix and the matrix of random shifts, all addresses can be derived in VHDL by:

```
addra_out <= conv_std_logic_vector(dv*((conv_integer(RS(xa/p,ya))
+(xa mod p))mod p)  ,addrwidth) + base(xa/p,ya);
```

The parameter p must be a power of two, which is usually no design limitation. The two matrices base and RS are stored using distributed RAM, because each of them requires only a quite little amount of memory.

3.3 Scheduling

The two presented code construction methods differ in the way to map the code to a parallel architecture. The approach of H. Zhong and T. Zhang requires more block memories, whereas my code construction method requires a scheduled access to the different memory blocks. This scheduled access should be described in this section.

A block diagram of the architecture can be found in Fig. A.11. Scheduling is realized in the interleaver modules (called address generator in the figure) and the two data multiplexers DI MUX and DO MUX. Reconsider the code example of Section 3.1.1, that can be mapped to a partly parallel architecture

with two (or four) units, since the derived constraint is fulfilled. The base matrix of the example is:

$$H_b = \left(\begin{array}{cccc|cccc} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{array}\right).$$

As all properties for mapping the code of the base matrix H_b also hold true for the expanded matrix, I will only consider mapping and scheduling of H_b for the sake of simplicity here. While processing the CNs of row 1 and row 3 concurrently, RAM 1 is accessed once by the CNs of decoding unit 1 and three times by decoding unit 2, whereas RAM 2 is accessed three times by decoding unit 1 and once by decoding unit 2, leading to a total access of $d_c = 4$ times for each block memory. First we construct the matrix $\tilde{S}$, that holds the memory accesses for RAM 1 (1) and RAM 2 (2) for all check nodes (rows):

$$\tilde{S} = \begin{pmatrix} 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 \\ 1 & 1 & 1 & 2 \\ 1 & 2 & 2 & 2 \end{pmatrix}.$$

The different columns indicate the accesses, i.e. column 1 the first access,..., column d_c the d_c^{th} access. Row 1 and 3 are processed concurrently, as well as row 2 and 4. A '1' in row 1 indicates an access to RAM 1, therefore row 3 must hold a 2 in the same column. The same must be done with row 2 and 4. This rearranging in $\tilde{S}$ produces the *scheduling matrix* S:

$$S = \begin{pmatrix} 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 \end{pmatrix}.$$

Now CN 1 accesses memory 1 while CN 3 accesses memory 2 first, then the memories are accessed the other way around. Hence data of RAM 1 is given to decoding unit 1 while data of RAM 2 is given to decoding unit 2 in the first cycle, then the data is exchanged in the remaining $d_c - 1$ cycles. This data exchanging takes place in DI MUX and DO MUX, and requires the knowledge of S.

In addition to that the address generation in the interleaver module has to be modified. In this example first the address for memory 2 has to be generated for CN 3, then the others for accessing memory 1. Hence the elements of the matrix I_b must be rearranged in the same way as in the derivation of

S from $\tilde{S}$. Moreover this rearrangement has accomplished in RS. In the architecture shown in Fig. A.11 each decoding unit is equipped with its own interleaver, generating the addresses for the message memory of this decoding unit. The messages of the second half of the columns are written to RAM 2, which reuses the addresses space of RAM 1. This leads the matrices $\tilde{I}_b$ (before rearranging its elements) and I_b (after rearranging its elements) of the example:

$$\tilde{I}_b = \begin{pmatrix} 2 & 0 & 2 & 4 \\ 0 & 3 & 4 & 3 \\ 1 & 5 & 6 & 6 \\ 7 & 1 & 5 & 7 \end{pmatrix}, \text{ and } I_b = \begin{pmatrix} 2 & 0 & 2 & 4 \\ 0 & 3 & 4 & 3 \\ 6 & 1 & 5 & 6 \\ 1 & 5 & 7 & 7 \end{pmatrix}.$$

Each interleaver only needs to access the message memory of its corresponding decoding unit, thus interleaver 1 only needs to generate the addresses of I_{b1} whereas interleaver 2 only needs to generate the addresses of I_{b2}. Thus the interleaver matrices can be derived by using the scheduling matrix S and the interleaver matrix I_b:

$$I_{b1} = \begin{pmatrix} 2 & 1 & 5 & 6 \\ 0 & 3 & 4 & 7 \end{pmatrix}, I_{b2} = \begin{pmatrix} 6 & 0 & 2 & 4 \\ 1 & 5 & 7 & 3 \end{pmatrix}.$$

These matrices are the ones needed to generate the addresses for the single decoding units. They are packed into an array consisting of $[I_{b1}, \ldots, I_{bpar}]$ and are stored in the VHDL package codedef.vhdl.

3.4 Automatic Code Construction

As well as for the serial architecture I automatically constructed the parity check matrix H, and all the required matrices (I_b,RS,S) in Matlab and automatically configured the templates of the VHDL modules. Again a VHDL package (called codedef.vhdl) is used to define the basic parameters and to get a scalable design. Finally the block RAMs are automatically configured by calling the Xilinx CoreGenerator from Matlab and passing the correct parameters.

The following routines are used:

- *[cb, Hb, baseldpc, base, ST_in] = createscheduleablebase(len/p, dv, dc, par)* returns a scheduleable base matrix H_b, and the two unsorted matrices *base* for interleaving and *ST_in* for scheduling. The construction

works with the *try and error* principle. A parity check matrix H_b is constructed randomly first, then I check, if the constraint for scheduling is fulfilled. Therefore I first derive the unsorted interleaver and scheduling matrices *base* and *ST_in* by doing modulo (length/par) operations and divisions on the interleaver of the randomly constructed matrix. The unsorted scheduling table (previously denoted by $\tilde{S}$) then holds the accesses of each check node to the different RAMs. If the number of accesses to each memory in all concurrently processed rows is equal to d_c, the constraint is fulfilled and the matrix returned. If not, this procedure is repeated.

- Next the base matrix is expanded by calling *[Hex, RS_in] = expandldpc(Hb, p, dc)*. In this routine every nonzero element of the base matrix is replaced by a random cyclic shift matrix of size $p \times p$, the shifts are saved in the matrix *RS_in*, which is the unsorted random shift matrix. Every zero of the base matrix is replaced by a zero matrix of size $p \times p$.
- Now the elements of the matrices *RS_in*, *ST_in* and *base* have to be rearranged as shown for $\tilde{S}$ and S, and for $\tilde{I}_b$ and I_b. This is again accomplished by using the *try and error* principle in the Matlab script *[ST, int, RS] = scheduleme(ST_in, base, par, RS_in)*. The algorithm rearranges *ST_in* row after row: A row is taken and all permutations of this row are produced. Then this permutations are compared with the previous, already sorted rows, corresponding to concurrently processed CNs, if there are coinciding elements for some columns. If so, the next permutation is taken, if not the current permutation is used as a sorted row. When the matrix *ST_in* is successfully converted to matrix *ST*, the elements of the other two matrices *base* and *RS_in* are rearranged the same way first, and secondly are rearranged in order to produce the matrices $[I_{b1}, I_{b2}, \ldots, I_{bpar}]$ and *RS*, as shown in Section 3.3.
- Finally the VHDL package is created by calling *createpackage(len, width, dv, dc, it, mode, corr, p, par, ST, int, RS)*. Mode again configures the postprocessing function (0=scalar, 1=offset, 2=no postprocessing), corr is the correction term for the offset function. The memory modules are instantiated by calling the Matlab routines *createmsgRAM(len/par, dv, width)* and *createioRAM(len/par, width)*, which create the required CoreGenerator input files and call the CoreGenerator in batch mode.

All these scripts can be called from a single Matlab script called *[cb, Hb, baseldpc, Hex, RS, sched_base, cex, exldpc, ST] = constructcode(len, dv, dc, width, p, it, mode, corr, par)*. This constructs a code with the desired parameters, for a desired amount of parallel units, configures the VHDL package for storing all required parameters and instantiates the memory modules using the CoreGenerator. Hence, by calling this routine a complete synthesiseable

VHDL design of the desired decoder is obtained, which immediately can be implemented on the FPGA.

3.5 Realization

For realizing the architecture, I reused several units of the parallel architectures, namely the variable and check node processor with the control unit. These three modules, equipped with a message RAM, form a decoding unit, as shown in Fig. A.11, and only need little adoptions for a use in the parallel architecture. Variable and check node processor again work pipelined, exploiting the dual port feature of the memories.

3.5.1 I/O Interface

The channel data has to be directed to the different decoding units, as all units hold their own I/O memories in the variable node processors. While data is received at the I/O interface, the first $length/par$ bits are passed to decoding unit 1, the second $length/par$ bits to decoding unit 2, ..., the bits $length/par \cdot (par-1)+1$ to $length$ are passed to the last decoding unit. $Length$ denotes the block length and par the number of parallel units. While streaming a block from the FPGA, this has to be done the other way around. Then the first $length/par$ bits are read from decoding unit 1, the second $length/par$ bits from decoding unit 2 ..., the bits $length/par \cdot (par-1)+1$ to $length$ are read from the last decoding unit. This is realized in the channel multiplexer. A ChipScope sample of a block transmission to the FPGA for a length 32 code with 4 parallel decoding units is shown in Fig. A.12. In the serial architecture the variable node processor signaled to the control unit that it received all values, but since all variable node units receive their data sequentially, this is not possible anymore if all units should start working simultaneously. Hence, the signal indicating that all data is transfered and processing can start (proc) is driven by the channel multiplexer. It is connected to all decoding units, which then start processing simultaneously.

3.5.2 Decoding Units

The modules of the serial architecture only need to be slightly adapted in order to reuse them in the parallel architecture. As already mentioned, the signal indicating that a block is transfered, is driven by the channel multiplexer. Timing in the check node processor has to be renewed, as the data read

and written to the message memory is delayed by an additional cycle due to the data multiplexers DI MUX and DO MUX. The interleaver now is not addressed anymore, but a signal for starting the address generation process is used.

The instantiation of the decoding unit is achieved by using the VHDL *generate* command, the decoding units are declared as components (ldpcdec):

```
G1: for I in 0 to (par-1) generate
dec_unit: ldpcdec
port map(
  lmclk      => mclk,
  channel_in => schannelmv((I+1)*width-1 downto I*width),
  channel_out => schannelvm((I+1)*width-1 downto I*width),
  ch_in_act  => sch_in_actmv(I),
  ch_out_act => sch_out_actmv(I),
  addraint  => saddraint((I+1)*addrwidth-1 downto I*addrwidth),
  addrbint  => saddrbint((I+1)*addrwidth-1 downto I*addrwidth),
  start     => sstart(I),
  proc      => sproc,
  DI_out    => sDI_in((I+1)*width-1 downto I*width),
  DI_in     => sDI_out((I+1)*width-1 downto I*width),
  DO_out    => sDO_in((I+1)*width-1 downto I*width),
  DO_in     => sDO_out((I+1)*width-1 downto I*width)
);
end generate;
```

Fig. A.15 shows processing of a $3, 6$ length 32 code in two parallel units over several iterations. DI_in[ix] and DO_in[ix] denote the bus connecting processing unit i with the DI MUX and DO MUX *in* ports, DI_out[ix] and DO_out[ix] the bus connecting DI MUX and DO MUX with processing unit i respectively. By comparing DI_in[ix] with DI_out[ix] and DO_in[ix] with DO_out[ix] carefully, the data exchange between decoding unit 1 and decoding unit 2 can be observed.

3.5.3 Interleaver

Each check node processor needs to access the message memory through its own interleaver, which is realized outside the decoding units as shown in Fig. A.11, but can be thought to be inside as well. Each can only derive the addresses of the needed memory, as explained in Section 3.3. The interleaver does not need to be addressed anymore, as it generates the addresses from

$I_{bi}, i = [1, par]$. The starting signal for generating the addresses is given by decoding unit 1 for all interleavers, as shown in Fig. A.15. On the other hand, the interleavers activate the DI MUX and DO MUX modules with the signals act_DI and act_DO respectively.

The interleaver modules are instantiated using the VHDL *generate* statement, the correct interleaving table is passed by using the VHDL *generic* command. The interleaver matrix ($base0$) and the random shift table RS are saved in the VHDL package as an array of the specific interleaver matrices I_{bi}, with $i = [1, par]$. The instantiation is accomplished in VHDL with (interleaveme is the interleaver component):

```
G2: for I in 0 to (par-1) generate
interleaver: interleaveme
generic map(
  base      => base0(I),
  RS        =>  RS0(I))
Port map(
  start     => sstart(I),
  addra_out => saddraint((I+1)*addrwidth-1 downto I*addrwidth),
  addrb_out => saddrbint((I+1)*addrwidth-1 downto I*addrwidth),
  act_DO    => sact_DO_int(I),
  act_DI    => sact_DI_int(I),
  clk       => mclk
 );
 end generate;
```

3.5.4 DI MUX, DO MUX

The DI MUX and DO MUX module, realize the data exchange of different message RAMs and CN processors - the scheduling. As this elongates the access time to the memory by one cycle, timing in check node processing has to be adapted. On the other hand the time to fill the pipeline stays constant, because the interleaver can start generating addresses already one cycle earlier.

The DI MUX (data in, see Fig. A.11) multiplexes all data buses for writing the results to the message RAMs, the DO MUX multiplexes the data buses for reading values. The number of ports is $par \cdot width$ for ports of mode *in*, as well as for ports of mode *out*, with *par* denoting the number of parallel units. Data from the *in* buses is passed to the *out* buses according to the scheduling matrix S (called ST in the realization). This is realized in VHDL as:

```
DI_out(width*(conv_integer(ST(conv_integer(row_cntr)+
   +k*VN_nb*dv/(dc*p),conv_integer(col_cntr)))+1)-1
   downto width*(conv_integer(ST(conv_integer(row_cntr)
   +k*VN_nb*dv/(dc*p), conv_integer(col_cntr)))))
   <= DI_in((k+1)*width-1 downto k*width);
DO_out((k+1)*width-1 downto k*width) <=
   DO_in(width*(conv_integer(ST(conv_integer(row_cntr)+
   +k*VN_nb*dv/(dc*p), conv_integer(col_cntr)))+1)-1
   downto width*(conv_integer(ST(conv_integer(row_cntr)+
   k*VN_nb*dv/(dc*p), conv_integer(col_cntr)))));
```

3.6 Results

Throughput

By using several parallel units, throughput increases almost proportionally to the number of parallel units. Each unit works on N/par variable nodes, and on $N/par \cdot d_v/d_c = M/par$ check nodes. Thus the total processing time of one block T_b for a length N code with par parallel units is equal to the processing time of a length N/par code in a serial architecture:

$$T_b = \left(\underbrace{(d_v \cdot N/par) + (d_v + 2))}_{\text{VN processing}} + \underbrace{(d_c \cdot M/par + (d_c + 3))}_{\text{CN processing}} \right) \cdot It + \left(\underbrace{(d_v \cdot N/par) + (d_v + 2)}_{\text{result output}} \right), \tag{3.3}$$

with It denoting the number of iterations.

Thus a (3,6) length 4096 LDPC code processed in 4 parallel units using 10 iterations consumes $(3 \cdot 1024 + 5 + 512 \cdot 6 + 9)10 + (3 \cdot 1024 + 5) = 64657$ cycles

Table 3.1: Throughput for $(3, 6)$ codes with different lengths processed in different architectures (clock frequency = 50MHz).

	serial arch.	parallel arch. (par=4)	parallel arch. (par=8)
N=1024	792,5Kbit/s	3,15Mbit/s	6,23Mbit/s
N=4096	793,2Kbit/s	3,17Mbit/s	6,32Mbit/s

for processing one block. This leads to a throughput of $4096 \cdot 50 \cdot 10^6/64657 = 3.17$Mbit/s. Throughput can only be further increased by either using more parallel units, a higher clock frequency or less iterations. Table 3.1 shows the throughputs of different architectures when using a length 1024 and a length 4096 code. When using a length 4096 code the throughput is slightly higher, because pipelines do not have to be filled that often. By increasing the number of parallel units, throughput increases almost proportionally, a slight loss due to additional time caused by pipeline filling can be observed again.

Below the resource consumption for this example implemented on the Spartan 3 FPGA and on an Virtex II Pro device can be found.

Resource Consumption

The consumed resources of a $(3, 6)$ length 2048 and a $(3, 6)$ length 4096 LDPC code, using 4 parallel units and an internal message width of 8 bits, are shown in Table 3.2. The critical paths in the design are 6.5ns and 6.9ns for the length 2048 and 4096 code respectively, leading to maximum clock frequencies of 153,8 MHz and 145MHz.

Resource consumption of the design when using $(3, 6)$ codes with lengths 4096 and 8192 processed in 4 parallel units with an internal message width of 4 bits can be found in Table 3.3. For all constructed codes a base matrix H_b of size (8×16) was considered. The critical paths of the design are 5.03ns and 4.56ns for the length 4096 and 8192 code respectively, leading to maximum clock frequencies of 199MHz and 219MHz.

In order to allow a comparison with other realizations on today commonly used FPGAs, I also synthesized the same decoders on a Xilinx Virtex II Pro (2vp30fg676-5) FPGA (Table 3.4. Comparing the results with the results of the serial architecture, it can be seen that throughput increases about

Table 3.2: Resource consumption of $(3, 6)$ length 2048 and 4096 LDPC codes, using four parallel units and an internal message width=8 bits.

	total available	Code $N = 2048$	Code $N = 4096$
Slices	3584	3148 (87%)	3201 (90%)
Slice Flip Flops	7168	2007 (27%)	2042 (28%)
4 input LUTs	7168	5532 (77%)	5630 (78%)
block RAMs	16	8 (50%)	12 (75%)

Table 3.3: Resource consumption of $(3, 6)$ length 4096 and 8192 LDPC codes, using four parallel units and an internal message width=4 bits.

	total available	Code $N = 4096$	Code $N = 8192$
Slices	3584	2665 (74%)	2692 (75%)
Slice Flip Flops	7168	1589 (21%)	1589 (22%)
4 input LUTs	7168	4697 (65%)	4763 (66%)
block RAMs	16	8 (50%)	12 (75%)

Table 3.4: Resource consumption of $(3, 6)$ length 2048, 4096 and 8192 LDPC codes, using internal widths of 4 and 8 bits and using four parallel units, synthesized on a Virtex II Pro FPGA.

width = 8bits	total available	Code $N = 2048$	Code $N = 4096$
Slices	3696	3065 (22%)	3096 (22%)
Slice Flip Flops	27392	2007 (7%)	2042 (7%)
4 input LUTs	27392	5532 (20%)	5630 (20%)
block RAMs	136	8 (5%)	12 (8%)
width = 4bits	total available	Code $N = 4096$	Code $N = 8192$
Slices	13696	2589 (18%)	2632 (19%)
Slice Flip Flops	27392	1554 (5%)	1589 (5%)
4 input LUTs	27392	4697 (17%)	4763 (17%)
block RAMs	136	8 (5%)	12 (8%)

proportionally with the amount of parallel units. The bottleneck regarding available resources in the serial architecture was the block RAM, limiting the usable block size. In the parallel architecture larger block lengths can be realized, as the interleaver does not have to be stored anymore. On the other hand, due to the derivation of the interleaver and the scheduling which takes place in DI MUX and DO MUX, additional slices and LUTs are required. Moreover CLB consumption due to several parallel decoding units increases about proportionally with the amount of parallel units.

The design allows several trade offs of all these factors: By decreasing the internal width memory and logic can be saved, but with the disadvantage of less decoding performance. Longer block lengths again increase the decoding performance, but then more block RAM is used. In addition to that throughput and resource consumption can be compromised, by choosing the amount of parallel units.

Conclusion

Two different architectures of LDPC decoders have been realized on an FPGA. First, a serial architecture was proposed, and problems with memory access conflicts when parallelizing the system, and memory consumptions of the interleaver when using randomly constructed codes were shown. Thus, a developed code construction method which allows scheduling in parallel architectures and deterministic derivation of the parity check matrix on the one hand, and addressing decoding performance on the other hand was introduced. Moreover the modifications of the modules used in the serial architecture were shown in order to reuse them in the parallel architecture.

Both implemented architectures only consume $2d_v$ cycles for processing one bit for one iteration in one decoding unit, by efficiently exploiting the dual port structure of the memories on the FPGA and introducing a pipelined design. Both designs are implemented scalable, which offers very fast reconfiguration of the decoder. This allows a rapid implementation of different architectures and examination of the available trade offs. Matlab scripts were written which construct a parity check matrix with desired parameters (such as blocklength, variable and check node degree) that can be mapped to an architecture with specified properties, e.g. number of parallel units, internally used bus widths or amount of iterations used. In order to reconfigure the design with desired parameters, only one Matlab script has to be called, which generates a parity check matrix and configures all required VHDL modules accordingly. The modules can be immediately synthesized and implemented on the FPGA, and a working hardware realization is accomplished.

A Additional Figures and Files

A.1 Serial Architecture

VHDL package

```
package codedef is

constant ioaddrwidth : integer := 11; -- Int[log2(length*2)]
constant addrwidth : integer := 12;   -- Int[log2(length*dv)]
constant VN_nb : integer := 1024;    -- number of var nodes
constant CHK_nb : integer := 512;     -- number of chk nodes
constant max_iterations : integer := 5; -- maximum number of iterations
constant width : integer := 8;        -- message width
constant dv : integer := 3;           -- variable node degree
constant logdv : integer:=2;          -- log of dv
constant dc : integer := 6;           -- check node degree
-- array definition for storing the messages while VN processing
type varmsg is array (dv-1 downto 0, 1 downto 0) of
std_logic_vector(width-1 + logdv downto 0);
-- array definition for storing the messages while CN processing
type chkmsg is array (dc-1 downto 0, 1 downto 0)
of std_logic_vector(width-1 downto 0);

  -- returns max value if input > 2**(width-1)-1, min value
  -- if input <-2**(width-1), and else returns input
function Saturate(
input: in std_logic_vector)
return std_logic_vector;

-- returns log2 (width) of an integer, e.g: log2(7)=3, log2(8)=4.
function Log2(
input: in integer)
return integer;
```

```
-- postprocessing function
function postproc(
input: in std_logic_vector)
return std_logic_vector;
end codedef;
```

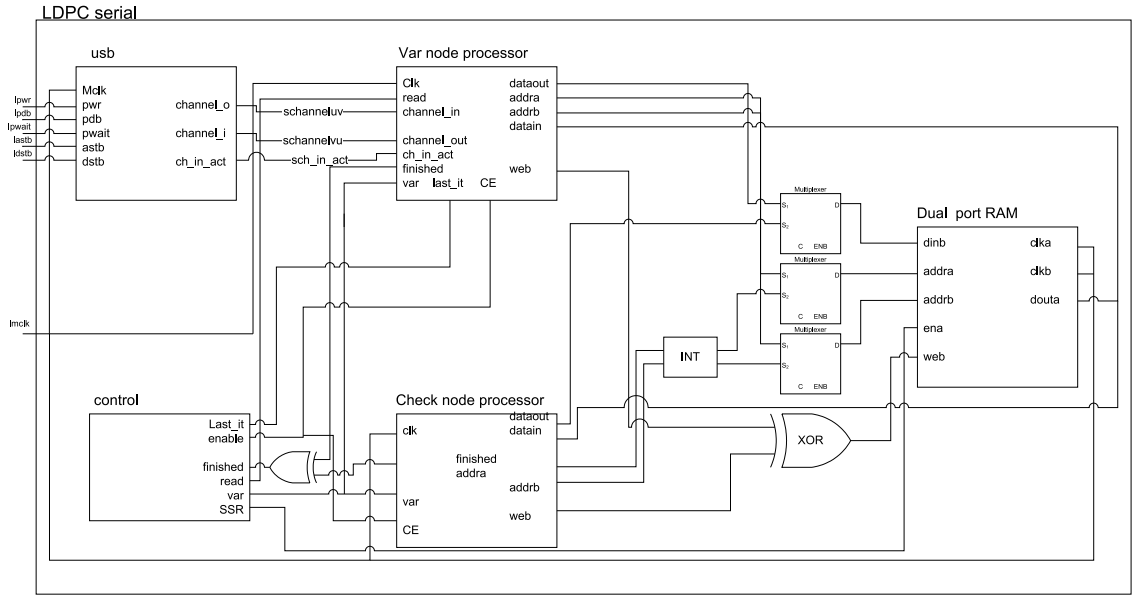

Figure A.1: Block Diagram serial Architecture.

A.2 Parallel Architecture

VHDL package

```
package codedef is
  constant par : integer := 4; -- number of parallel units
  constant ioaddrwidth : integer := 10; -- Int[log2(length*2)]
  constant addrwidth : integer := 11; -- Int[log2(length*dv)]
  constant VN_nb : integer := 512;  -- number of var nodes
  constant CHK_nb : integer := 256;  -- number of chk nodes
  constant max_iterations : integer := 5;
  constant width : integer := 8;
  constant dv : integer := 3;
```

```
  constant logdv : integer:=2;
  constant dc : integer := 6;
  constant p : integer := 128;
  type varmsg is array (dv-1 downto 0, 1 downto 0) of
    std_logic_vector(width-1 + logdv downto 0);
  type chkmsg is array (dc-1 downto 0, 1 downto 0) of
    std_logic_vector(width-1 downto 0);
  type schedTab is array (0 to 7  , 0 to dc-1)of
    std_logic_vector(1 downto 0);
  type baseinter is array (0 to 1,  0 to dc-1) of
    std_logic_vector(10 downto 0);
  type randomshift is array (0 to 1  , 0 to dc-1) of
    std_logic_vector(6 downto 0);
  type defbaseinter is array (0 to 3) of baseinter;
  type defrandomshift is array (0 to 3)  of randomshift;

constant base0: defbaseinter:=((("00000000000","01100000000",
"00000000001", "00000000010","10010000001","10010000000"),
("01100000001","00110000001","00110000000","00110000010",
  "10010000010","01100000010")),
(("00000000010","01100000010","00110000000","00000000001",
  "00110000001","10010000001"),
("10010000010","00000000000","10010000000","00110000010",
  "01100000000","01100000001")),
(("00000000001","00110000000","00110000010","01100000000",
  "01100000001","10010000001"),
("00110000001","00000000010","01100000010","00000000000",
  "10010000000","10010000010")),
(("00000000000","10010000000","00000000010","10010000001",
  "00110000000","01100000000"),
("00000000001","01100000001","00110000010","01100000010",
  "10010000010","00110000001")));
constant RS0: defrandomshift:=((("1110001","1110010",
  "0000100","0111101","0100101","0001001"),
("1000010","0100000","0100101","1010000","1110100","1111110")),
(("0110011","1011110","0101101","0110111","1101000","0001001"),
("0101101","1000100","0011000","0010011","1011011","1110011")),
(("0110000","1110011","0010101","1001101","1100000","1111111"),
("0001001","1110111","1110011","1100011","1000100","0110011")),
(("1011100","0011111","1000011","1100010","1110101","0101100"),
("0111000","1100100","0011011","1010111","1110000","0010101")));
constant ST: schedTab:=(("00","00","01","10","11","11"),
("00","01","01","10","10","11"),
```

```
("10","10","00","01","01","00"),
("10","11","00","01","01","00"),
("11","11","10","00","00","10"),
("11","00","11","11","00","01"),
("01","01","11","11","10","01"),
("01","10","10","00","11","10"));

  -- returns max value if input > 2**width,
  -- min value if input <-2**width, and else returns input
function Saturate(
input: in std_logic_vector)
return std_logic_vector;

-- returns log2 (width) of an integer, e.g: log2(7)=3, log2(8)=4.
function Log2(
input: in integer)
return integer;

-- postprocessing function
function postproc(
input: in std_logic_vector)
return std_logic_vector;
end codedef;
```

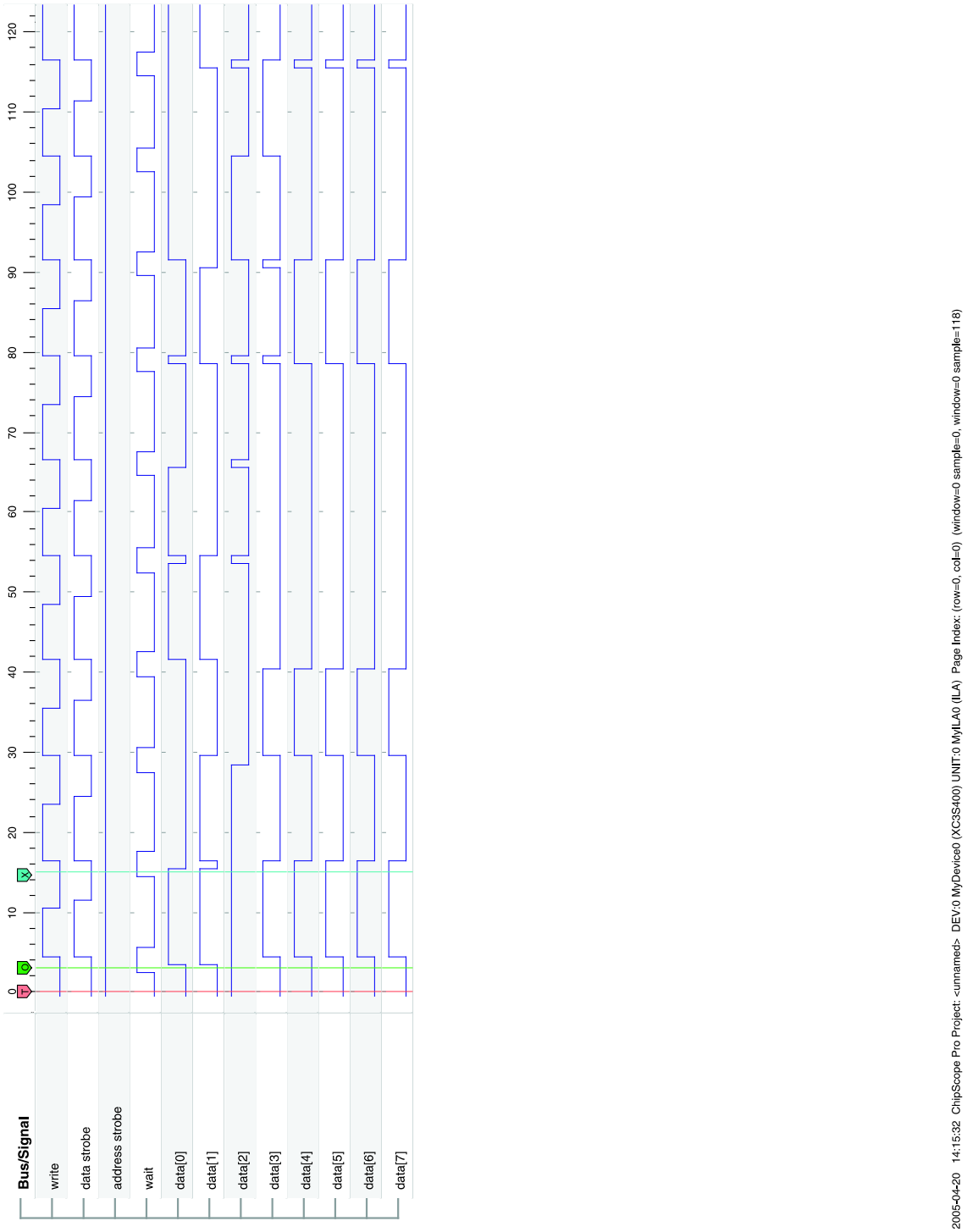

Figure A.2: USB interface signals.

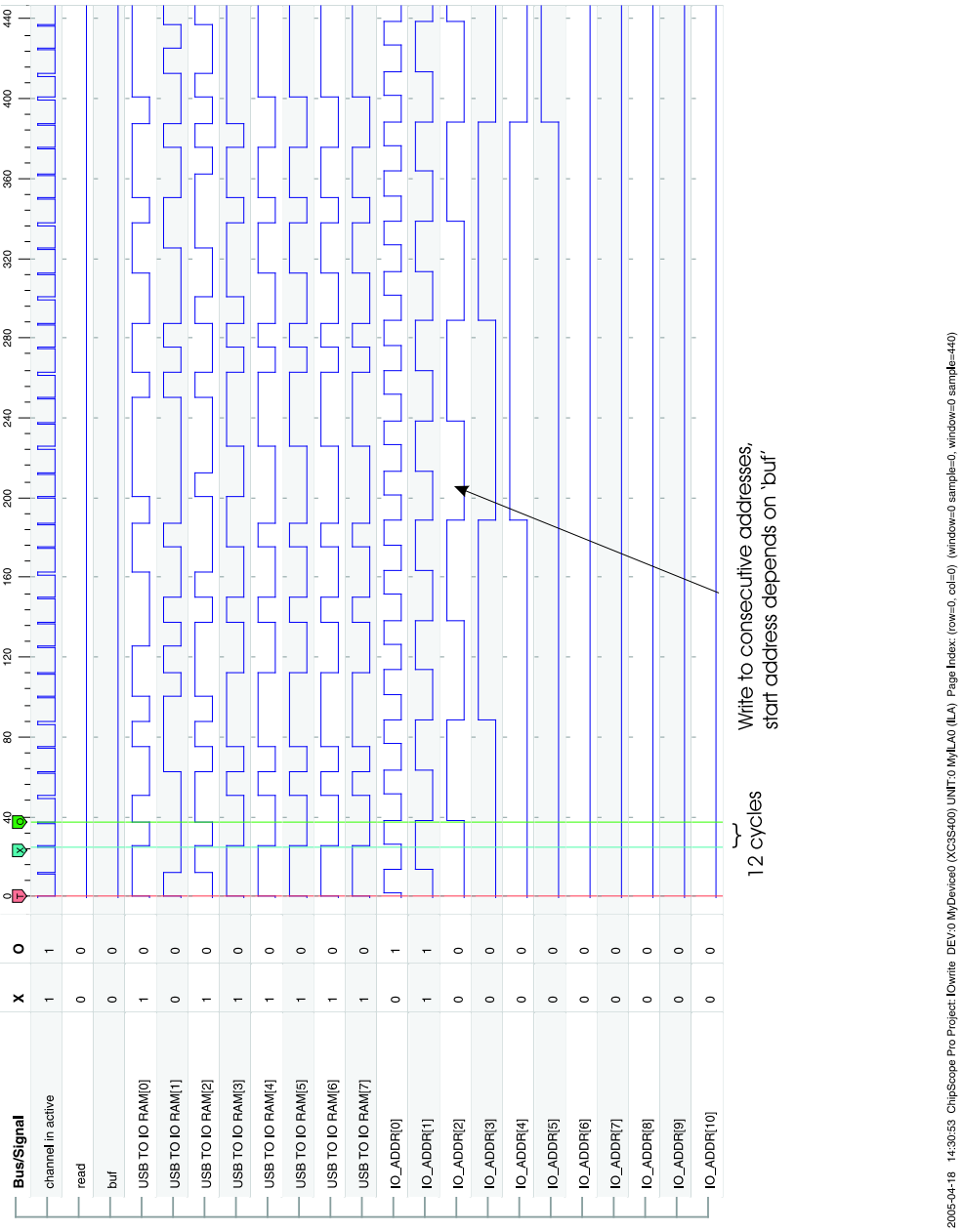

Figure A.3: Interface: block write.

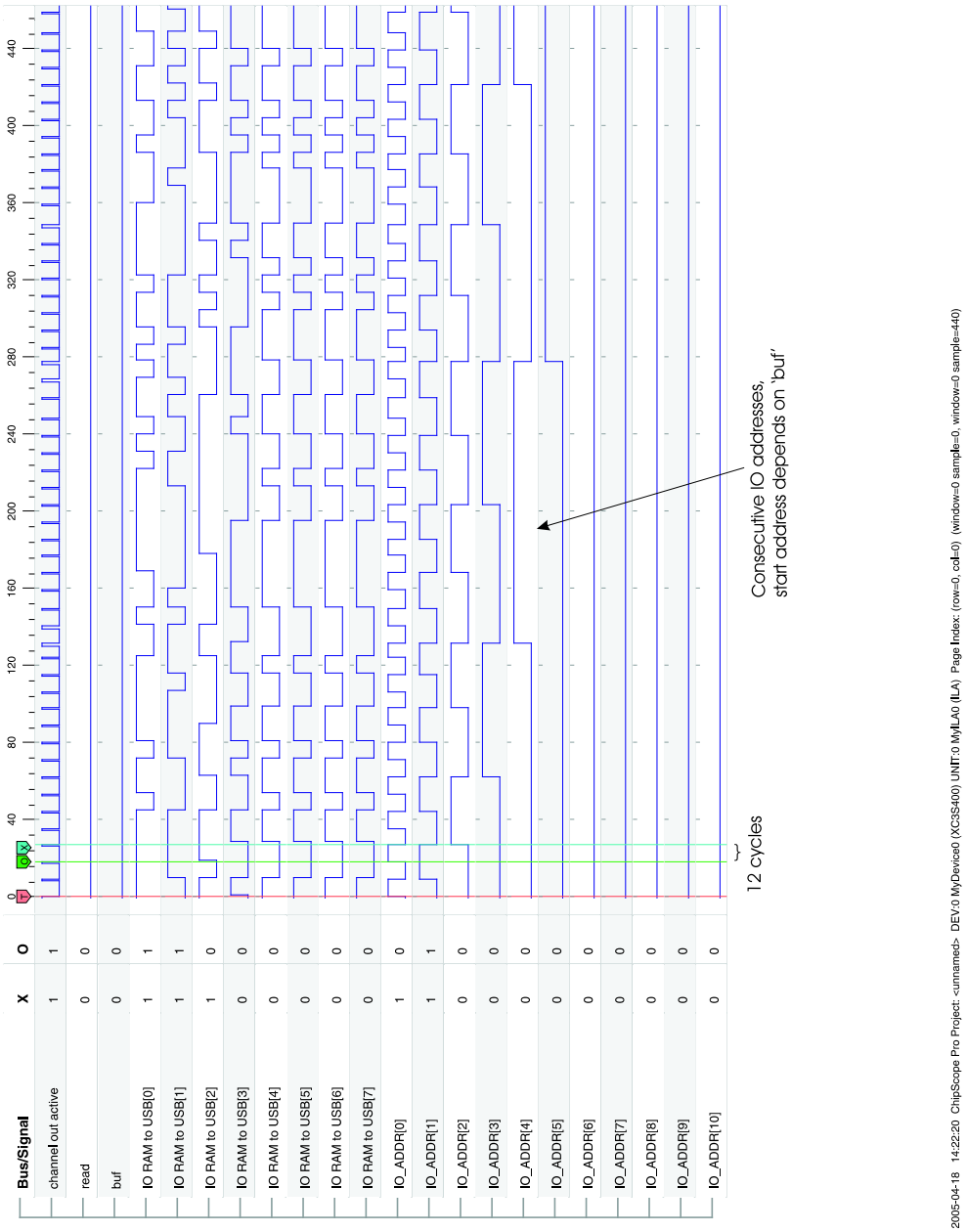

Figure A.4: Interface: block read.

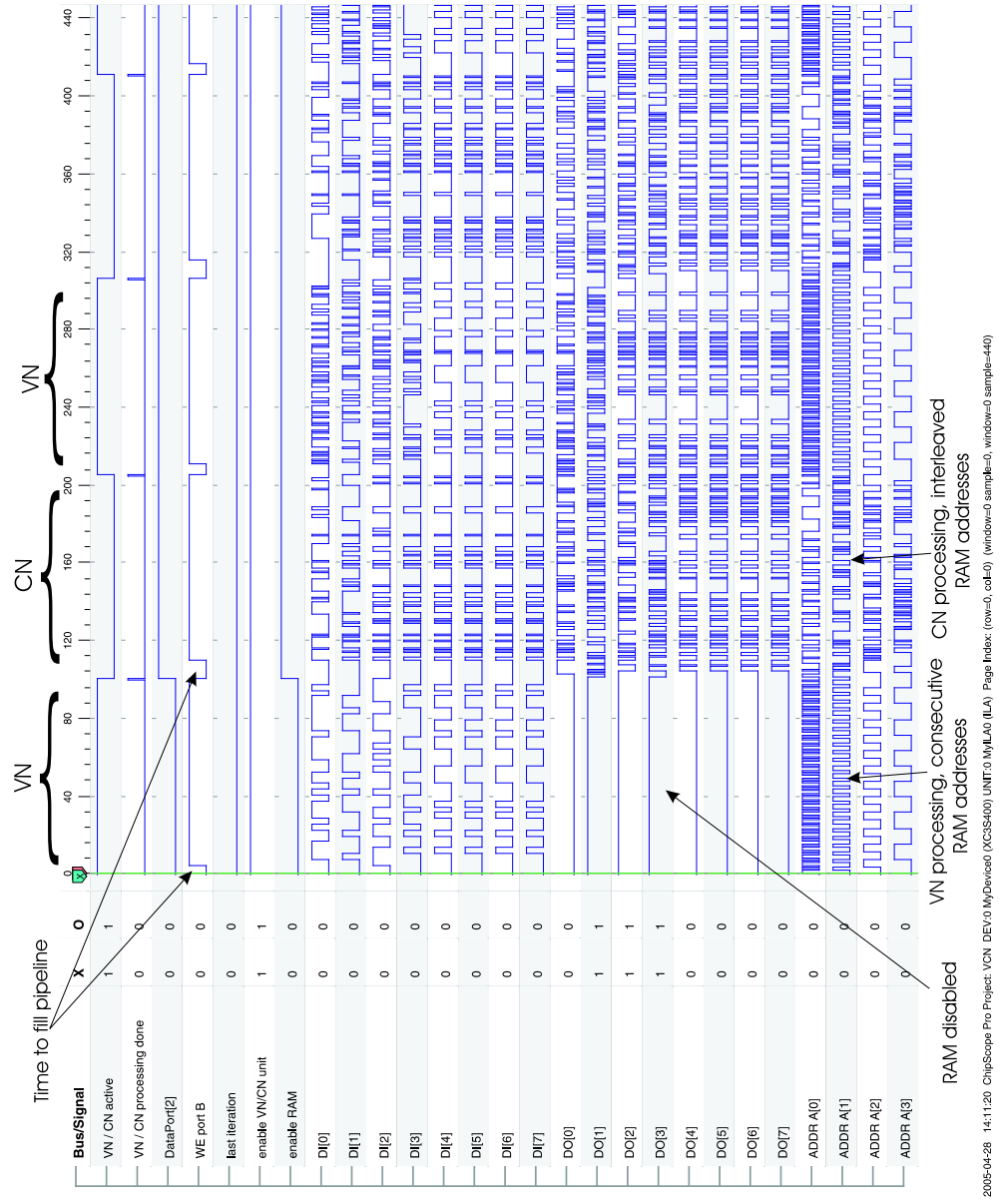

Figure A.5: Variable and Check node processing (p.1/2).

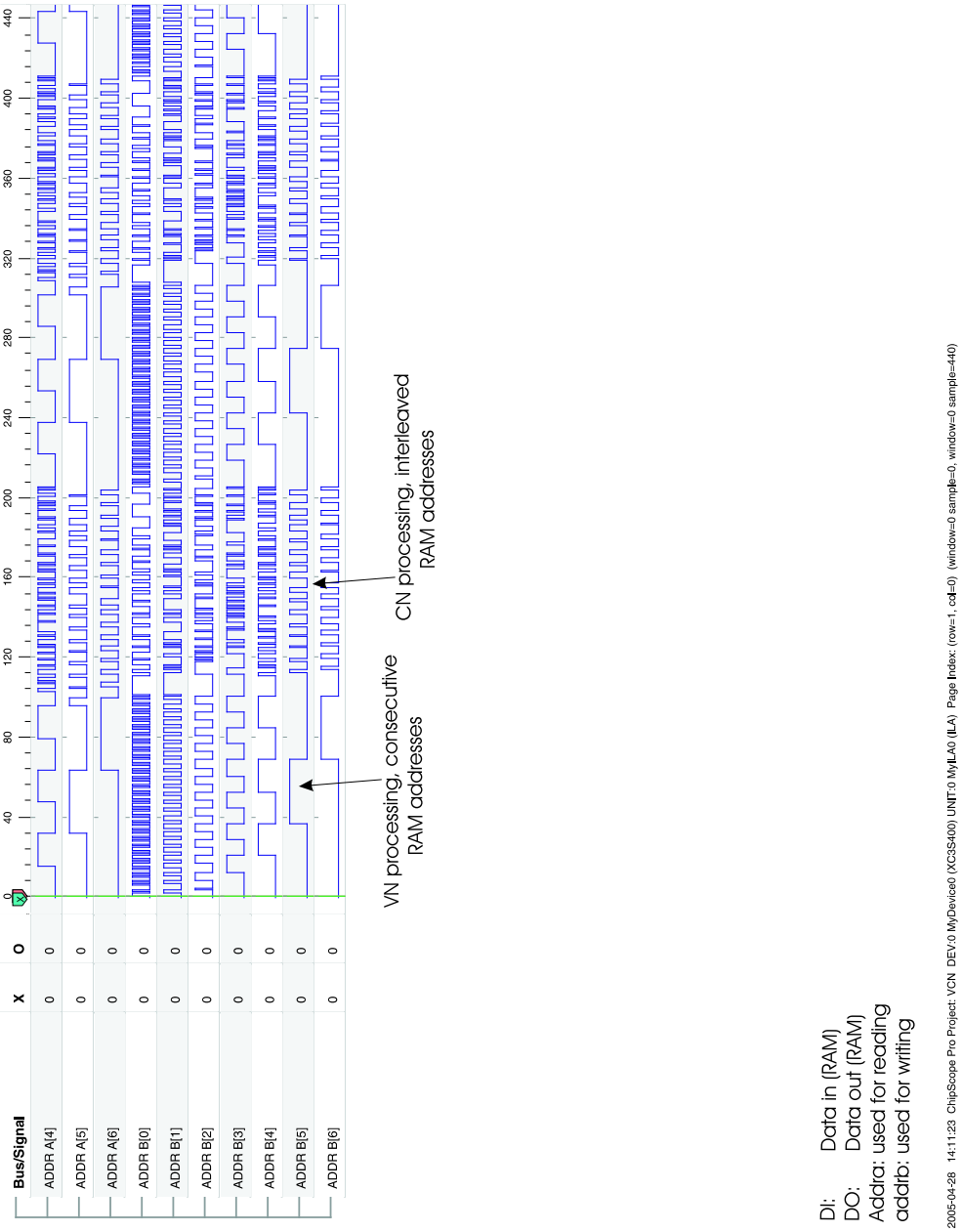

Figure A.6: Variable and Check node processing (p.2/2).

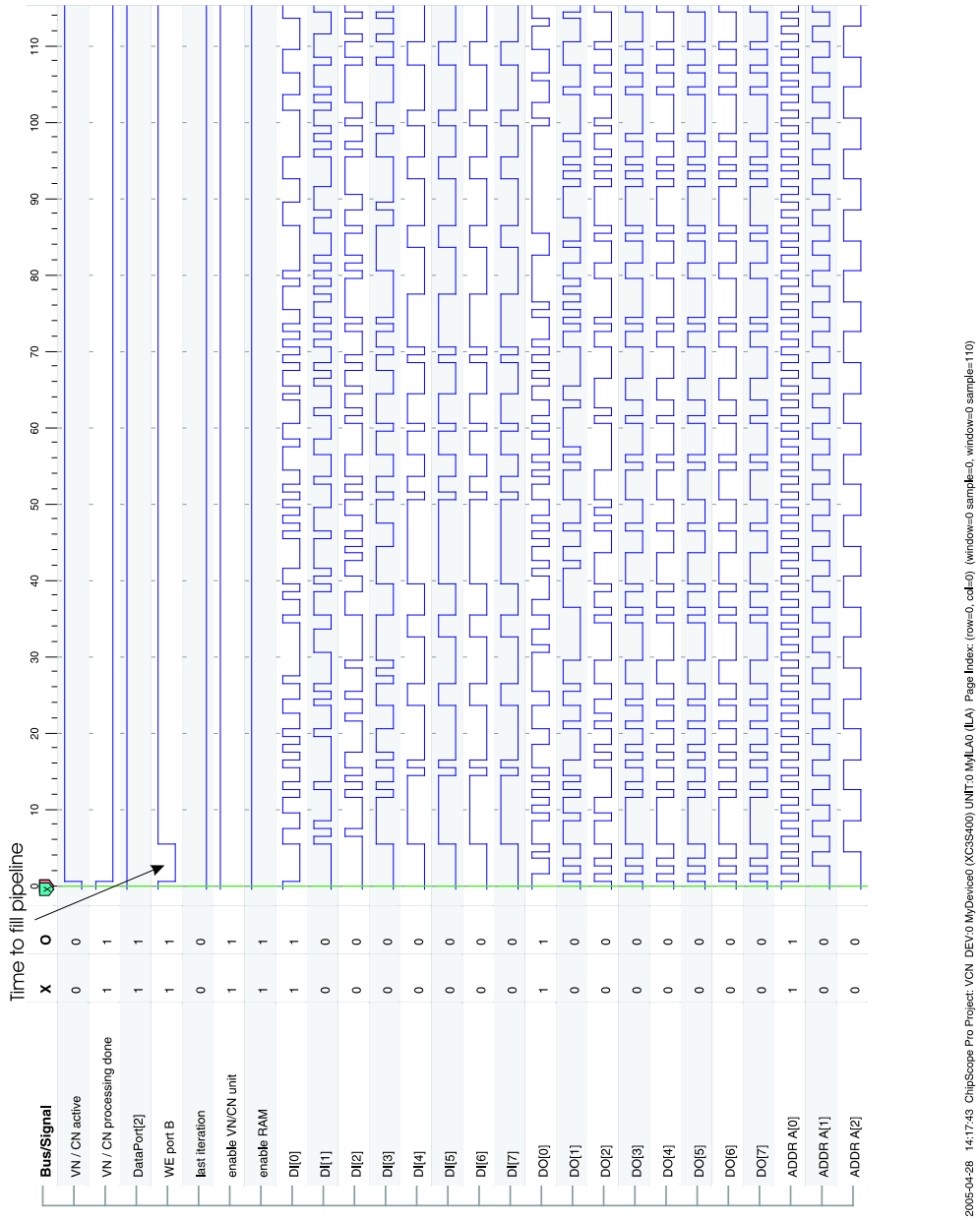

Figure A.7: Pipelined Variable node processing (p.1/2).

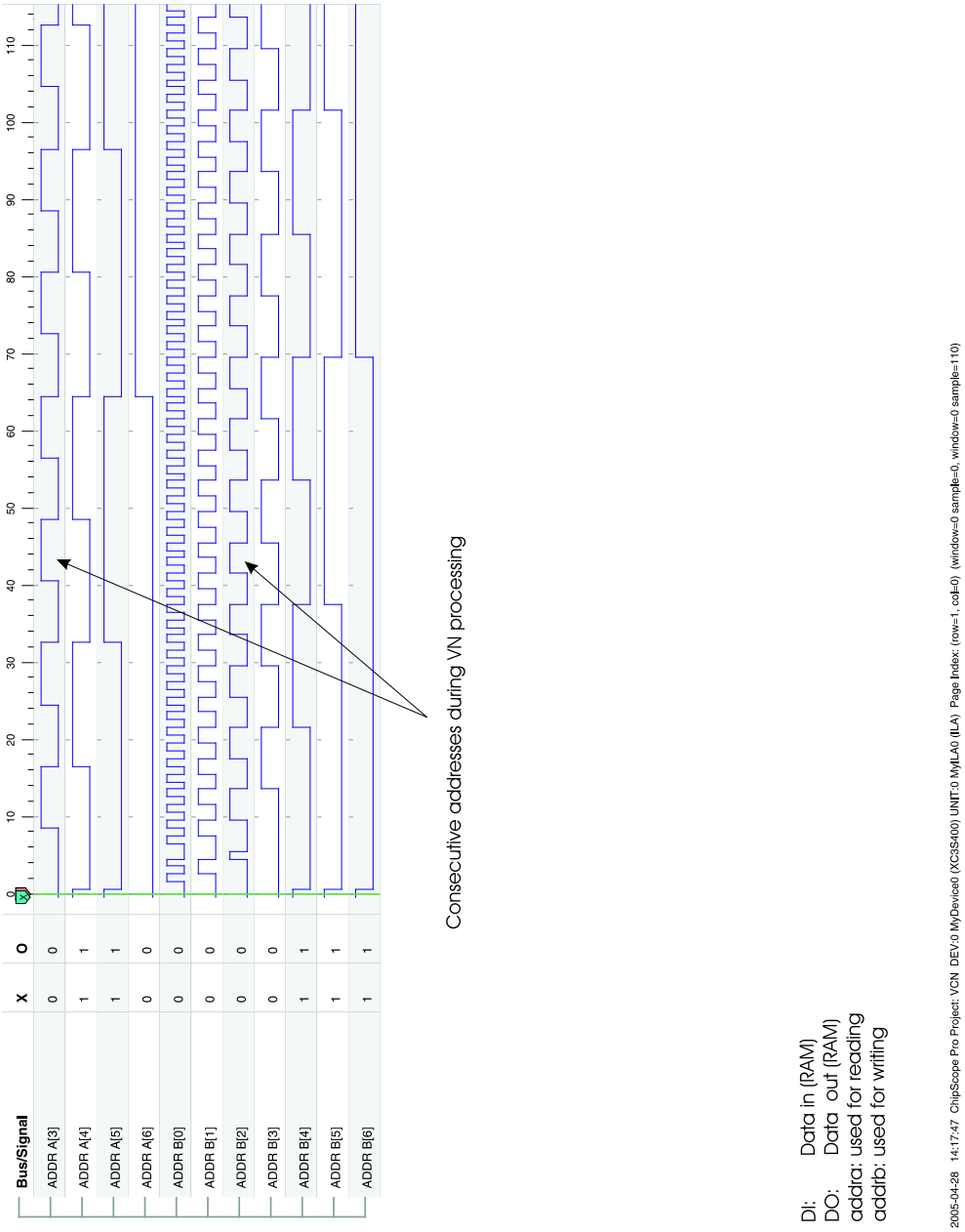

Figure A.8: Pipelined Variable node processing (p.2/2).

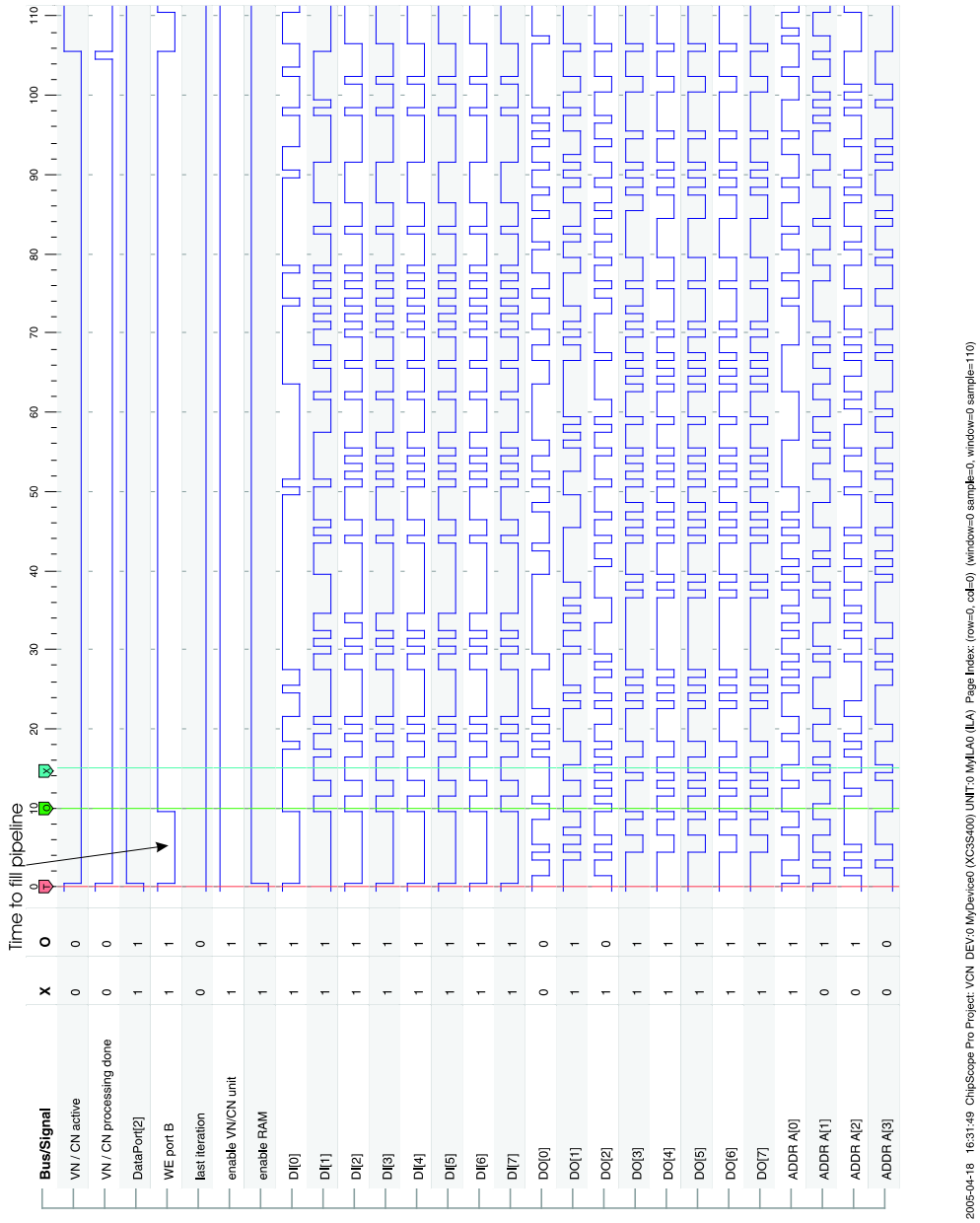

Figure A.9: Pipelined Check node processing (p.1/2).

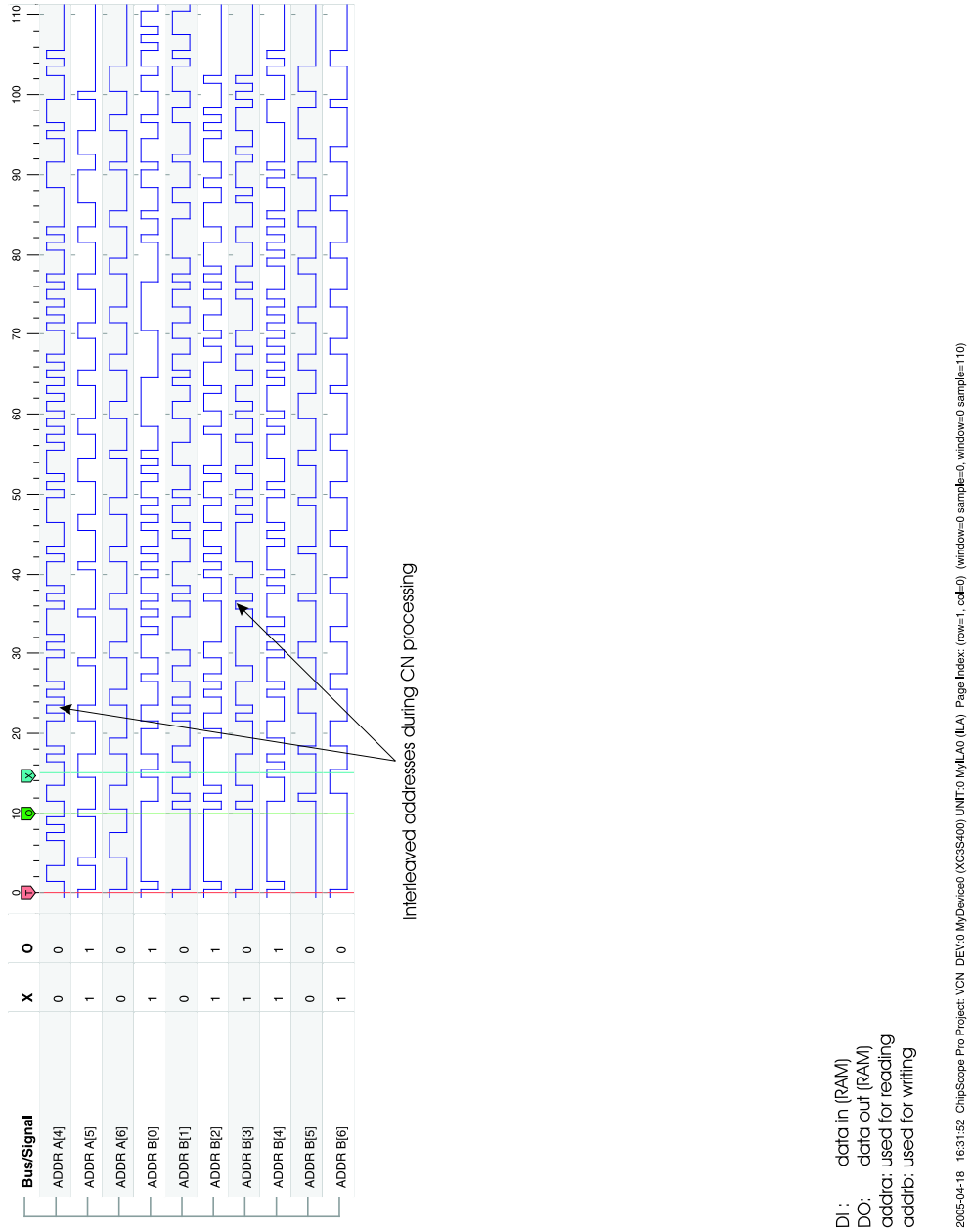

Figure A.10: Pipelined Check node processing (p.2/2).

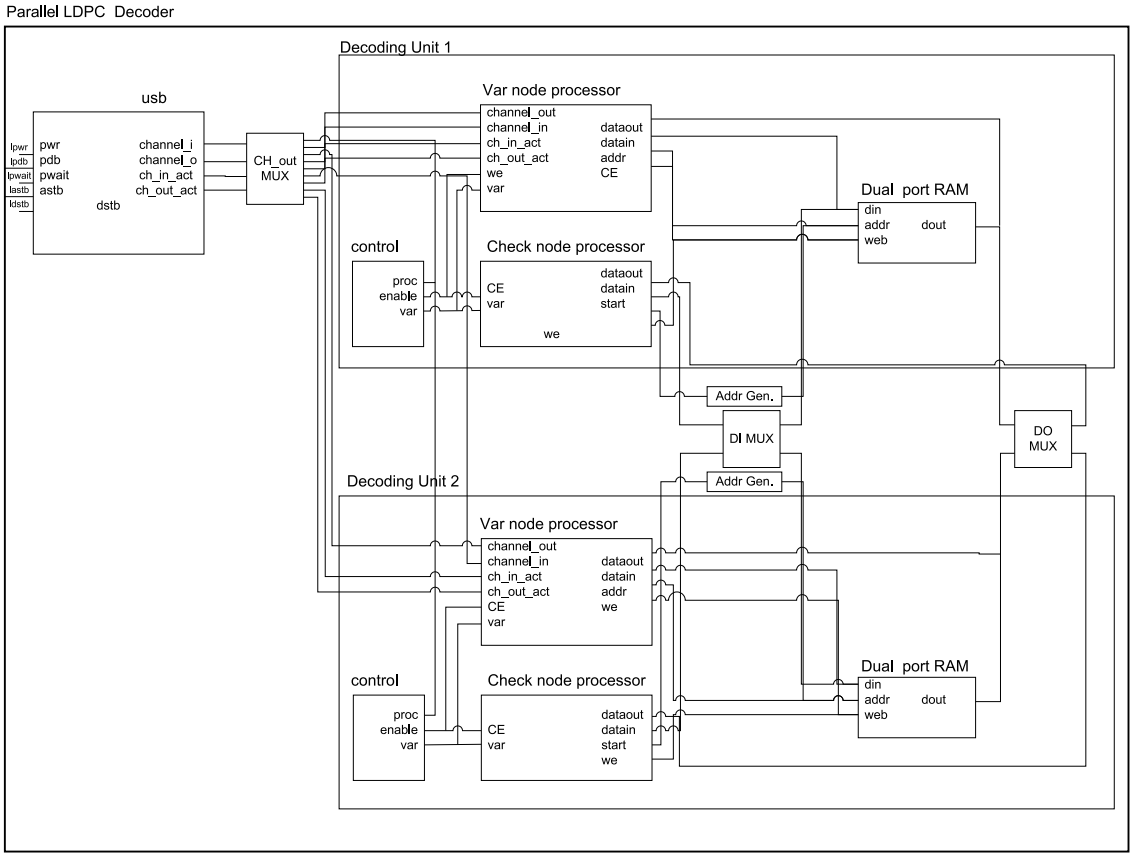

Figure A.11: Block Diagram parallel Architecture

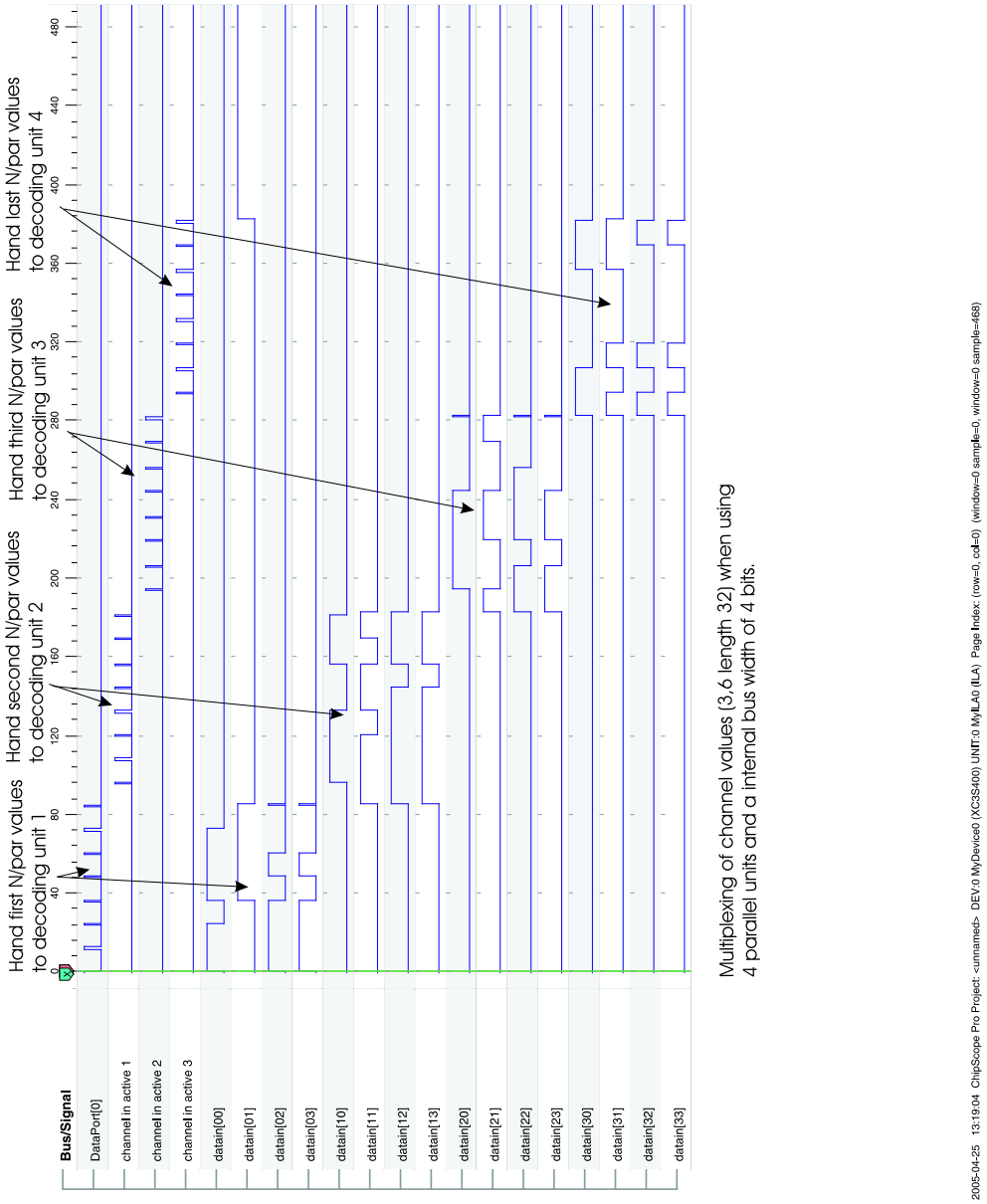

Figure A.12: Channel MUX, write cycle

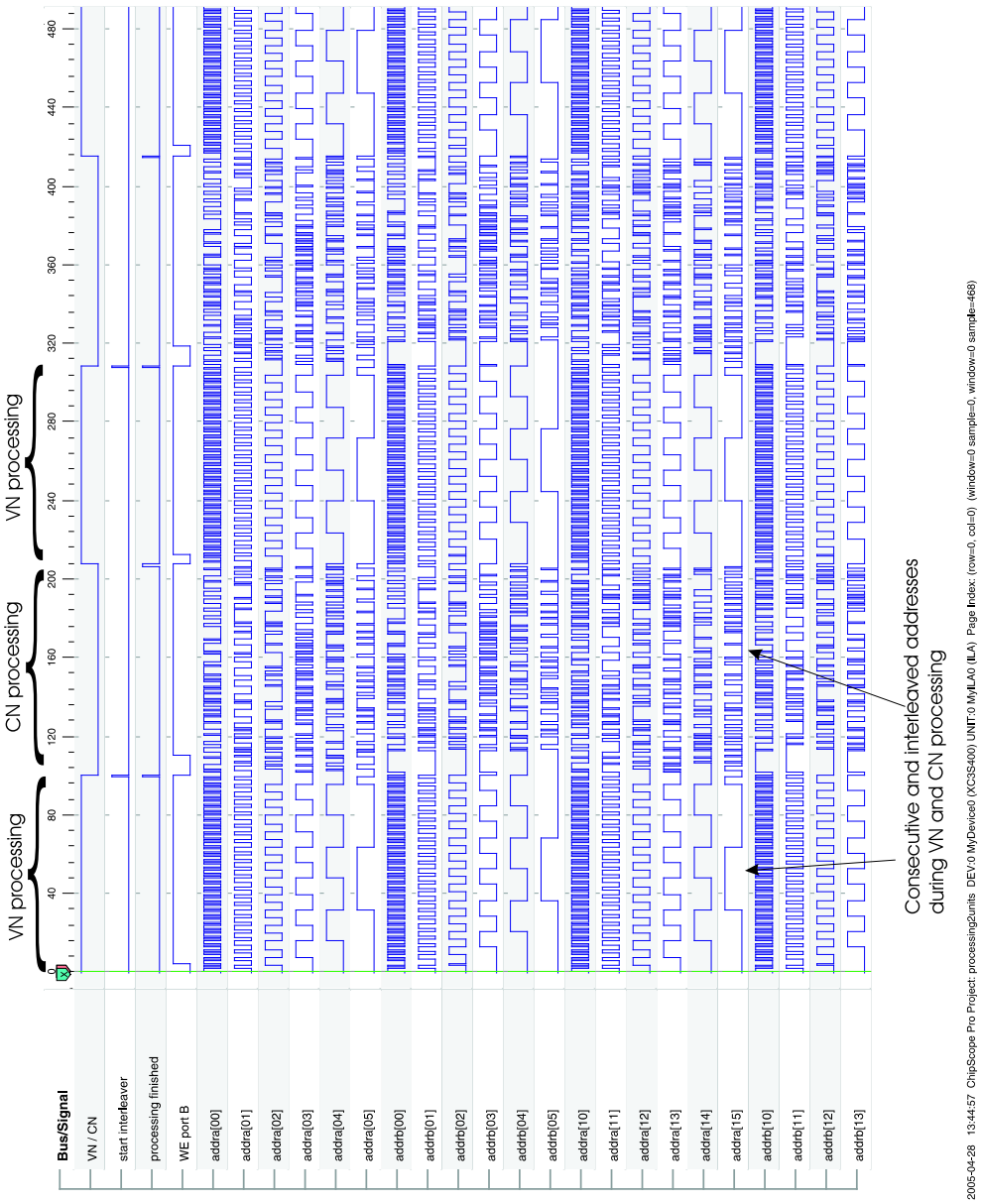

Figure A.13: VN and CN processing in parallel Architecture (p.1/3)

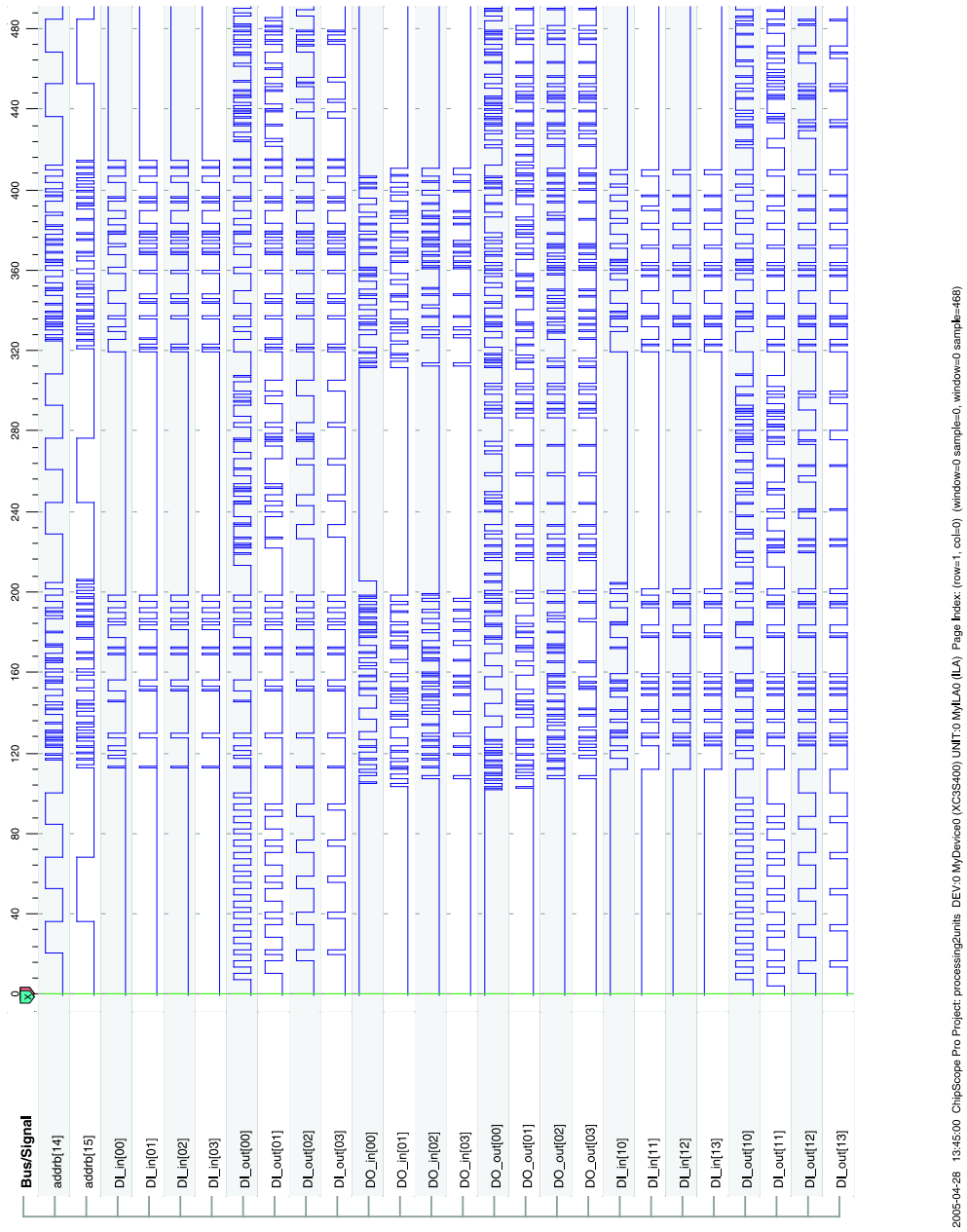

Figure A.14: VN and CN processing in parallel Architecture (p.2/3)

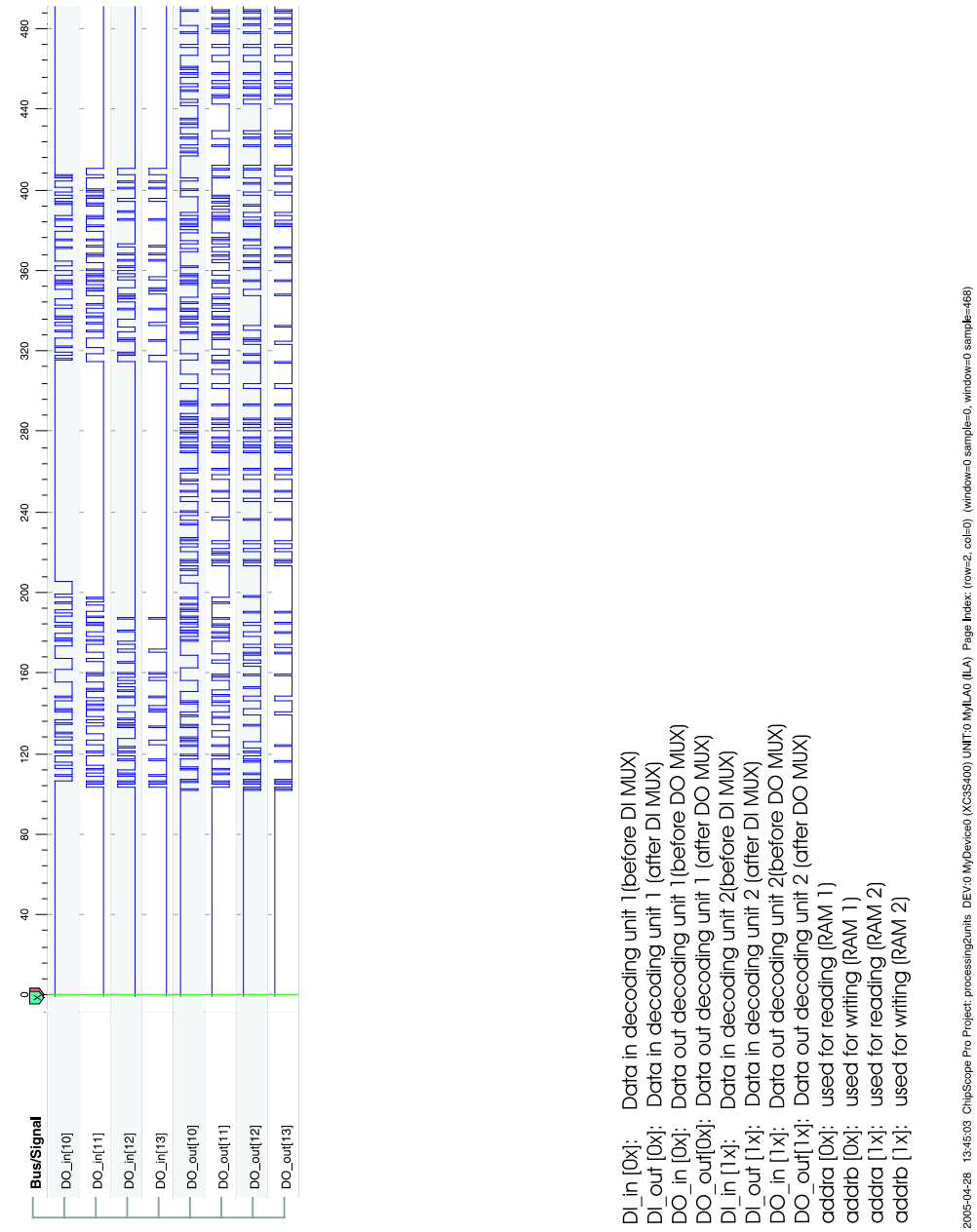

Figure A.15: VN and CN processing in parallel Architecture (p.3/3)

B List of Abbreviations

LDPC	Low Density Parity Check
SPA	Sum Product Algorithm
SMA	Sum Min Algorithm
BP algorithm	Belief Propagation Algorithm
CN	Check node
VN	Variable node
LLR	Log Likelihood Ratio
BIAWGN	Binary Input Additive White Gaussian Noise
AWGN	Additive White Gaussian Noise
VHDL	Very high scale integration Hardware Description Language
USB	Universal Serial Bus
RAM	Random Access Memory
ROM	Read Only Memory
I/O	Input Output
FPGA	Field Programmable Gate Array
LUT	Look Up Table
CLB	Configurable Logic Block
ISE	Integrated Software Environment
JTAG	Joint Test Action Group
PROM	Programmable Read Only Memory
LED	Light Emitting Diode
API	Application Program Interface
BER	Bit Error Rate

Bibliography

[1] R. G. Gallager, "Low Density Parity Check Codes", *M.I.T Press*, 1963.

[2] D.J.C.MacKay, "Good error-correcting codes based on very sparse matrices", *IEEE Transactions on Information Theory*, vol.45, pp.399-431, Mar. 1999.

[3] G. Lechner, J. Sayir, "Improved Sum Min Decoding of LDPC codes", *ISITA 2004, international symposium on information theory and its applications*, Parma, Italy, 2004.

[4] J. Chen, M. P. Fossier, "Density evolution for two improved BP-based decoding algorithms of LDPC codes", *IEEE communications letters*, Vol. 6, No. 5, May 2002, pp.208-210

[5] A.J. Blanksby and C.J. Howland, "A 690mW 1Gb/s 1024-b, rate 1/2 low-density parity-check code decoder", *IEEE Journal of Solid-State Circuits*, vol. 37, no.3, pp. 404-412, March 2002

[6] Digilent Inc., "dpimref.vhd", available at: http://digilent.us/Materials/referencedesigns.html

[7] Digilent Inc., "dpimref.pdf", available at: http://digilent.us/Data/VHDLSource/EPP_Parallel/DpimRef.pdf

[8] Digilent Inc., "Digilent Port Communications Programmers Reference Manual", available at: https://www.digilentinc.com/Sales/software.html

[9] Xilinx, "Spartan-3 FPGA Family: Complete Data Sheet", available at: http://direct.xilinx.com/bvdocs/publications/ds099.pdf

[10] Digilent Inc., "Spartan-3 Starter Kit Board User Guide", available at: https://www.digilentinc.com/Data/Products/S3BOARD/S3BOARD-rm.pdf

[11] Digilent Inc., "Digilent USB 2 Board", available at: http://www.digilent.us/Data/Products/USB1/usb2-brochure.pdf

[12] Digilent Inc., "Digilent USB 2 Module Reference Manual", available at: http://www.digilent.us/Data/Products/USB1/USB2_RM.pdf

[13] Digilent Inc., "Digilent Port Communications Programmers Reference Manual", Sept. 21st 2004, available at: https://www.digilentinc.com/Sales/software.html

[14] Hao Zhong and Tong Zhang, "Design of VLSI Implementation-Oriented LDPC Codes", Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY, USA

[15] Xilinx, "Xilinx ISE 6 Software Manuals and Help - pdf collection", available at: http://toolbox.xilinx.com/docsan/xilinx6/books/manuals.pdf

[16] Xilinx, "XST User Guide", available at: http://toolbox.xilinx.com/docsan/xilinx6/books/docs/xst/xst.pdf

[17] Xilinx, "ChipScope Pro Software and Cores User Guide", July 30th, 2004