Universitat Politecnica de Catalunya
Barcelona

The implementation of an LDPC decoder in a Network on Chip environment

Author : Massimo Camatel
Tutor : Antoni Gelonch

thanks to : Ismael Gomez, Guido Masera, Fabrizio Vacca

In collaboration with
Politecnico di Torino

Index

1. Abstract

The proposed project takes origin from a cooperation initiative named NEWCOM++ among research groups to develop 3G wireless mobile system. This work, in particular, tries to focuse on the communication errors arising on a message signal characterized by working under WiMAX 802.16e standard. It will be shown how this last wireless generation protocol needs a specific flexible instrumentation and why an LDPC error correction code suitable in order to respect the quality restrictions. A chapter will be dedicated to describe, not from a mathematical point of view, the LDPC algorithm theory and how it can be graphically represented to better organize the decodification process.

The main objective of this work is to validate the PHAL-concept when addressing a complex and computationally intensive design like the LDPC encoder/decoder. The expected results should be both conceptual; identifying the lacks on the PHAL concept when addressing a real problem; and second to determine the overhead introduced by PHAL in the implementation of a LDPC decoder.

The mission is to build a NoC (Network on Chip) able to perform the same task of a general purpose processor, but in less time and with better efficiency, in terms of component flexibility and throughput. The single element of the network is a basic processor element (PE) formed by the union of two separated components: a special purpose processor ASIP, the responsible of the input data LDPC decoding, and the router component PHAL, checking incoming data packets and scanning the temporization of tasks execution.

Supported by a specific programming tool, the ASIP has been completely designed, from the architecture resources to the instruction set, through a language like C. Realized in this SystemC code and converted in VHDL language, it's been synthesized as to fit onto an FPGA of the Xilinx Virtex-5 family. Although the main purpose regards the making of an application as flexible as possible, a WiMAX-orientated LDPC implemented on a FPGA saves space and resources, choosing the one that best suits the project synthesis. This is because encoders and decoders will have to find room in the communication tools (e.g. modems) as best as possible.

The whole network scenary has been mounted through a Linux application, acting as a master element. The entire environment will require the use of VPI libraries and components able to manage the communication protocols and interfacing mechanisms.

2. Introduction

2.1 The Flexible Radio Concept

In the mid 80's, began the research of an increasingly efficient mobile communication technique, a phenomenon that continued to grow during the 90's. At the beginning there were only wired connections but the potential of wireless communication soon became evident and then, thanks to the advance of technology, the transmission of information via Wi-Fi became a reality.
Since then, the demand for ever newer and more complex services for the user, required the implementation of new wireless standards. And these new standards were designed to accomplish these service requirements, such as higher bit rate/bandwidth and even more strict QoS (Quality of Service) parameters than the established standards of GSM, GPRS or UMTS.

The new standards have a double purpose; the first objective is to offer better services than those of wired connections; the second one tries to overcome the WiFi limitations due to its technological nature by offering these services even in extreme difficult conditions.

A standard, like WiFi or WiMAX, not only has to establish some restrictions and parameters and manage services for a medium. A very important feature should be to satisfy a wide description of the equipment needed for installation. Therefore, the most recent standards require a *flexible* equipment, which must be able to adapt to different conditions that may occur during their operation.

2.1.1 The Definition of Flexibility

The definition of flexibility in a radio context is described by four major features that a device should have. These are the following:

- **adaptivity :** literally it's the radio system automatic response to some changes in the circuit behavior by the modification of the numerical value of a set of parameters. This adaptive reaction is provided by a control feedback over some change-status signals, constantly monitored, and then by a subsequent action of an arbitrator on a number of signals;
- **reconfigurability :** it's the ability of reconfigure at a structural or architectural level some of its modules without causing a significant change to the set of parameters;
- **modularity :** describes the degree of separation and recombination of components in a system. All components can be decomposed into smaller sub-systems, faciliting the process of planning, reviewing and debugging. This way you can also have a better control of the more complex block, just by making sure that the simpler blocks work correctly separatly;
- **scalability :** is the property to handle large amounts of work when required and to maximize by combining with hardware components.

All these features, or at least some of them, are necessary to deal with all the standard requests and any inconvenience that may arise from the environment.
In fact, every standard has different restrictions and parameters about location of the terminal, its speed if moving and the quality of the required service. Joining different features services implies using different standards. If we'd want to apply to different services which exploit different standards and we are employing a terminal carrying a single standard, we should change machine

whenever we must change service.

Furthermore, many services, especially as regards the most recent ones, such as video streaming and broadcasting or high-speed internet, exploit more than one standard simultaneously (e.g., WiFi for video, UMTS for voice). For this reason too, the equipment must implement several standards or a standard flexibility way of thinking.

For this reason, to ensure flexible working and the means to support the services is something must be done. Every component related to the WiMAX technology must implement a flexible architecture or, such as in our case, a flexible processing algorithm, that could work with differentt codes or in different working conditions.

Finally, the speed rate of innovations in wireless standards requires an improvement of equipment capacity, through the upgrading of new software and various applications.

2.2 The WiMAX standard

In the new generation standard, we focused on the WiMAX, a technology for transmitting wireless network, based on the IEEE 802.16 standard. This is not a new technology, besides is a more innovative and commercially viable adaptation of a technology already in use to deliver broadband wireless services in private facilities around the globe.

It's very important for the components making part of the WiMAX equipment to respect the flexibility characteristic, since this standard will provide many new services in the most extreme conditions, like, as it'll be shown, the high-speed mobil connection (for a standard extension) and the connections in presence of particular kind of obstacles. In addition to that, since the nature of WiMAX wireless communication system is based on broadband signals, the connection of several users and lots of working transactions will be performed, increasing network traffic and requiring an efficient routing device.

WiMAX standard has the advantage of allowing a wireless connection between a Base Transceiver Station (BTS) and thousands of receiver points without requiring a direct line of sight (NLOS, Non Line Of Sight). Actually, WiMAX can overcome only small barriers, such as trees or buildings, although the supply of bandwidth turns out to be limited to 20 Mbit/s. It will never be able to overcome hills or mountains.

It can be used within a wide range of plane territories such as metropolitan areas, where it could reach the 70 Mbit/s throughput. Indeed, when the standard was developped it was thought that the transmission of broadband could reach the distance of 50 Km. But the amount of tests carried out revealed that, at the frequency of 3.5 GHz, the performance was acceptable only through few kilometres, before it started to lose efficiency. Neverthless, it's important to remember that the transmission power used (the EIRP, Equivalent Isotropically Radiated Power) during the experiments was limited to the value of 36 dBm, i.e. 4 Watts. In any case the network's coverage can be considered efficient enough.

2.2.1 The WiMAX main features

The main advantages making the WiMAX technology a very competitive hope for the foreseeable future in the wireless communication environment, can be sumed up as:

- **flexibility :** as the definition of flexibility assures, it's the ability of adaptation of a device or environment against a change of the whole system conditions. Or in other words, the ability to work under different working conditions or parameters;

- **security :** provides models of encryption, authentication services and security guarantees;
- **quality of service :** it's measured. The QoS parameters are latency, jitter and packet loss numerical coefficients. The WiMAX presents very low latency across the wireless span (many has less then 10 milliseconds), but also jitter tolerance and low percentage of packet loss;
- **throughput :** the IEEE 802.16 modulation system can transport a large amount of network traffic with high spectrum efficiency and excellent tolerance to reflected signals;
- **installation :** only a simple antenna is required for the basic equipment;
- **interoperability :** since WiMAX is a standard, it is independent from the apparatus or the provider;
- **mobility :** because of the 802.16e standard, the connections are available even on mobile environment at 120 Km/h;
- **cost/coverage ratio :** the economy scale of components production, in addition to the open character of the standard allows the cost reduction with the implementation of a high-speed bandwidth;
- **NLOS (not line of sight) :** initially, the particular modulation performed in the WiMAX standard was used to create a wireless connection between two points that are not exactly having a direct contact sight, but within a parcially obstructed environment. Actually, some experiments revealed the decrease of bandwidth and performance: only with the advent of the 802.16e standard, the efficiency in this exceptional situation has improved, though less than expected.

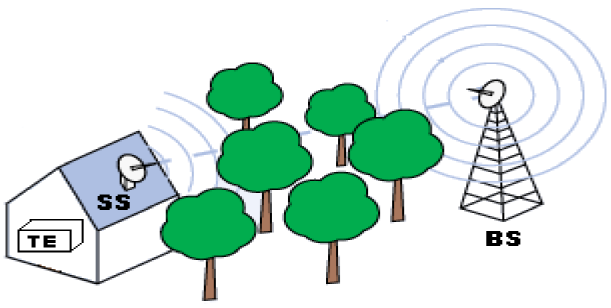
These and other potential of the WiMAX brought this new technology to be used in many other fields, such as Wi-Fi hotspots connection, xDSL technology and mobile phone high velocity services and connection.

2.2.2 The WiMAX physical system

A WiMAX 802.16 standard network is simply made of 3 elements:

- *Base Station (BS)* : this fixed wireless station receiver, amplifies and retransmits the signals coming from other stations faraway;
- *Subscriber Station (SS)* : it's connected to one Base Station, with which exchanges data signal;
- *Terminal Equipment (TE)* : it's the final apparatus, by which the user is connected to the network and is able to send/receive messages.

An example of how the WiMAX network works is shown in the picture below.



The Base Station represents the gateway allowing all the users connection to the WiMAX network: it provides coverage to all the Subscriber Stations in their area of influence and manages the traffic to and from these. The Terminal Equipment (e.g., a personal computer) is connected to the Subscriber Station to deliver it via packet traffic to the Base Station that will care to send it to the final recipient.

The biggest rival technologies, we may compare the WiMAX with, are the 3G systems, such as UMTS and CDMA2000. The WiMAX appears to be superior to the previously cited technologies in speed of transmission and range of cell coverage.
For all practical purposes, a conflict has no reason to exist because WiMAX is designed to connect distant points between them, with applications as external networks that have great coverage through licensed spectrum, while on the other hand Wi-Fi has a short range (like ten meters) with no licensed spectrum, suitable only for providing a good service for a local home network.

2.2.3 The 802.16e standard

The standard 802.16e was born in october 2005 to give the possibility of a mobile connection. According to the characteristics mentioned in the IEEE package documentation, this standard should help mantain the connection, while sending and receiving information, of a terminal moving up to a speed of 120 Km/h.
Basically they can be resumed as:

- **band coverage** of radio frequencies (2.3 GHz – 2.5 GHz – 3.3 GHz – 3.5 GHz – 3.8 GHz);
- a **multiple acces OFDM (Orthogonal Frecuency Division Multiplexing) modulation** splits the spaces of frecuency in several sub-carriers, reducing the interferences for terminals with omnidirectionals antennas;
- **occupation channels scalability** in order of disponibility of band;
- **better handover management**, i.e. the passage of a terminal from a base station to another holding the connection;
- **roaming management**.

Besides all the equipments and fixed stations already showed, the mobile 802.16e standard is provide of others apparatus, as:

- *Access Service Network Gateway (ASN GW)* : it's the network equipment assembled with all the base stations in the coverage area. It controls the authentication functionality, managing the QoS parameters, besides of dealing with the accounting procedures and holding the encryption keys. It's in charge of the mobile IP registration, so it's mandatory for a terminal moving through different base stations be furnished with this apparatus;
- *Home Agent Mobile IP (HA)* : it supports the mobility of a terminal through differents ASN GW and the IP address changing;
- *AAA server* : the authentication server AAA realizes the authentication of the user through username and password or smartcard to complete the connection operations.

All these features should allow a terminal on a moving vehicle to keep the connection, as we said, up to a speed of around 120 km/h, limited by the constraints of the *handover* protocol under the QoS (Quality of Service) range. Indeed, exceeded the speed limit, the power of signal and signal to noise ratio no longer meet the requirements of quality, resulting in the inmediate disconnection of the terminal.

This leads to an handicap of the WiMAX for a user traveling at a higher speed (like on a high-speed train); at least until the improvement of IEEE 802.16 standard parameters, such as technic of modulation, bandwidth and others features of transmissing channels.

2.2.4 The error correction system

An essential service for the transmission that must be considered and developed as best as possible is the error correction system. Through a wireless connection, it's likely that data is corrupted and the information received is different from the original broadcast. The channel introduces errors in the information bits and the WiMAX technology has to manage a system to correct as much errors as possible, because it's not able to control the communication channel. That's true for every kind of communication technologies already in use. But since WiMAX exploits services requiring a very high quality of service, it must implement a very efficientcoding/decoding error correction system.
For that reason, redundancy of no-information bits is introduced in the information words and at the time of transmitting and receiving a decoder block uses this redundancy to correct errors.

Hence, the standard suggests the use of the following coding methods:

- **Convolutional Code (CC);**
- **Convolutional Turbo Code (CTC);**
- **Low-Density Parity-Check (LDPC).**

CC is now considered obsolete for the applications requiring high performances, where the CTC and LDPC codes are used. However, the complexity of CTC and LDPC codes is much higher then the CC codes, which makes it suitable for small applications where benefits are not so strict parameters.

Within the category of high-performance codes, the LDPC is, without doubts, the best choise, since there are many advantages that lead over CTC. First of all, they are characterized by the need of a fewer amount of resources, for the same quality of encoding/decoding operations, and thus a smaller amount of area occupied. Furthermore, carrying large quantities of information, LDPC shows the greatest encoding codeword lenght, up to 2304 bits compared to 960 bits of the CTC.

The LDPC code is based on the use of SISO (Soft-Input Soft-Output) decoders, that differ in many aspects from the oldest ECC (Error-Correcting Control) decoders. Indeed, the ECC system implements a low complexity to achieve high code rate. SISO decoders have probabilistic inputs and outputs, rather than bit frames, which forces the use of digital signal processing (DSP) to obtain useful information, resulting in greater complexity; nevertheless, they gain in code rate over the ECCs.

The DSP used for the decoding operations implements a passing iterative algorithm, a powerful error correction method for an improvement in bit error rate. Basically, this type of algorithm receives an encoded input as a frame of bit with a certain amount of redundancy. Therefore, this is used to create some messages that are internally exchanged between two sets of processing elements, called *nodes*; these instruments should be implemented in a software or a hardware. The nodes are the representation of the raw and columns of the parity check matrix used to create the code word from the original message. The exchanging of messages and the parity control done by the nodes allow to correct the errors in the code word and to extract the original message.
Hardware implementation implies that the nodes would be single blocks able to receive messages

from the other ones, process the data and eventually return them to the other nodes. In addition, since the work of the two sets must be fully separated, there should be a control unit, as a FSM (Finite State Machine) controlling and determining the temporization.

2.2.5 Implementation Issues

There are several ways to implement the LDPC decoder. All these methodologies have their pros and cons. There are already several solutions to the problem of LDPC decoding, each one using different technologies and different approximations. As will be discussed in the concluding part, some devices of the state-of-the-art, thought reaching excellent performances, completely lost for what concerns the characteristic of flexibility, in this case regarded as the ability for the decoder to obtain the original information by working with different LDPC codes.

In the project carried out, the importance of the characteristic of flexibility for the LDPC decoder is likely to influence the choice of hardware platform. Thus a preferred software solution is not so much a question of resources available: the algorithm structure becomes suitable to changes of the LDPC code.

The software solution, that's been adopted in this project, consists in writing a code dedicated to implement the LDPC algorithm. But in this way, the resources involved would no longer be controlled, at least not with the normal high level programming languages, such as C or C++. However, through the use of particular types of languages, such as the SystemC programming language, this is made possible.

The processor chosen to perform the passing iterative algorithm of the LDPC code is a special instruction set one, hence able to perform only a specific program code. This so called ASIP, is not meant to be for general purpose use, since the instructions, although their assembler nature, are created ad-hoc. The ASIPs are used to improve performance for a very specific application.

By the SystemC language, which is a mixture between C and the assembler code, it's performed the implementation of a dedicated assembler instruction set into the ASIP processor, customized for our purposes, in a language like C. These instructions will be executed into the main code program of the processor, the one that will represent the algorithm. Not only the instructions are implemented by the user, but every other resource in the processor is declared there, in order to avoid wastage of memory area.

The LDPC algorithm created in the software way, implemented on the program main code of an ASIP approach solution, is the better choice in order of efficiency and performance. As the number of processant nodes is relevant, in case it meants to share the nodes between different ASIPs, it will be necessary to foresee some routing operation into the ASIP instruction set. Therefore, to avoid too much work to the ASIP and mantain it effective for the algorithm purpose, it should be associated with a component managing the communications with the outside world, the PHAL. The PHAL would help the ASIP to perform the Tanner graph node operations by synchronize the steps and managing the messages delivering.

The whole LDPC encoding system will be a network of Processor Elements (PE), formed by the union of ASIP and PHAL, a so called Network on Chip (NoC).

3 LDPC Concepts

3.1 Introduction

The LDPC codes (Low-Density Parity-Check) are particular linear corrector block codes, used to correct errors of bit information transmission through channels of communication.
The name means that the code is based on a little distribution of 1's in his parity check matrix H; in other words, this matrix has a very small number of 1's in each row and column respect to the number of 0's.

This type of code was invented in 1960 by Robert Gallager during his doctorate, but since it required a computation of great complexity and since in the same years the Reed-Solomon codes, considered very suitable for error control coding, came out, the LDPC code remained unused for long time. Only from 1998, thanks to MacKay and Richardson/Urbanke the potentiality of this instrument was rediscovered and it began to get used for control purposes.

3.2 Detailed Concepts

The main feature of the code is the H matrix; it's been said that it has very few 1's, so if it's possible to say that H is a matrix *nxm*, it has a number of Wc 1's per raws and Wr 0's per columns, where

$$Wc << n \qquad\qquad Wr << m$$

If the number of 1's in the columns and in the rows is constant, the code is regular, whether it's irregular if not.

An LDPC code is always managed starting from its H matrix, by which its manipulation it's possible to reach the generator matrix G, that permits the creation of coded informations. In fact, merging the H matrix, this is representable as

$$\boldsymbol{H = [P^T \mid I]}$$

and then it's easy to obtein the generator matrix G as

$$\boldsymbol{G = [I \mid P]}$$

Therefore, using this one, the coding phase of the LDPC code involves multiplying the original message for the matrix to obtein the coded message to send:

$$\boldsymbol{c = xG}$$

like all the corrector codes that add redundancy; *x* is the message to codify and *c* the encoded message to send. Since the encoding operation add some bits to better control the correction of errors, the encoded message c is bigger than the original one x; so if for example *x* is of 3 bits and the encoding adds 3 bits, making *c* of 6 bits, the code will be a *(6, 3)* one, ie with a ½ code rate.
In the presented case, the LDPC decode is performed with a (2304, 1152) code, so it can be said that the code doubles the number of bits of information, with the same type of ½ code rate.

Basically there are two manner to describe this kind of code; one is the previous matrix representation, that's quite heavy having to manage very big matrices, being a 1152x2304.
The other better way is a graphical representation. This is based upon a bipartite graph called *Tanner Graph*. The reason of the name is that the graph is organized in two sets of nodes and each node can be connected only to a node of the other set; the sets are

- **Variable Nodes (VN)** : N nodes that represent the N bits of a codeword;
- **Check Nodes (CN)** : M nodes that implement the parity-check calculation.

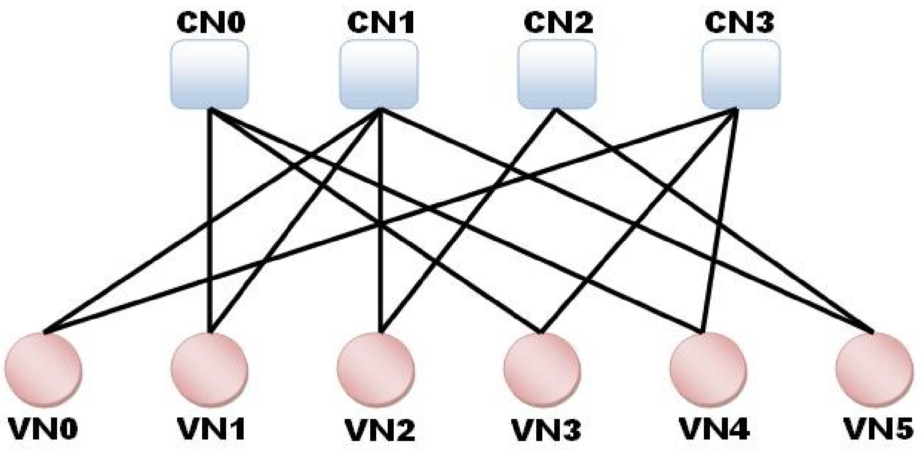
A VN node can be connected only with others CN nodes and a CN node only with others VN nodes. The connection among them and their number depends from the H matrix: in fact, there are *m* VN nodes, as the number of columns in the H matrix and *n* CN nodes, as the number of rows. There's a directly correspondence with the previous matrix representation because the previous (6, 3) code, for example, would be implemented through 6 Variable Nodes and 3 Check Nodes; so the same for ASIP approach to the LDPC decoding, will be implemented 2304 VN nodes and 1152 CN nodes.

A connection between the *i-th* VN node and the *j-th* CN node exists only if the bit in the H matrix at the *i-th* column and *j-th* raw is 1:

$$VN(i) \leftrightarrow CN(j) \qquad \text{if} \qquad H(i, j) = 1$$

Here it's possible to see an example of Tanner Graph, although it's not a perfect example of “low density” because of the little size of the H matrix, that should be larger to respect the definition.

$$H = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$



How it's shown, there are two groups of nodes; 4 CN nodes, as the number of rows and 6 VNs as the number of columns. And there's a connection between nodes CN and VN depending on the H matrix: CN0, representing the first row, is connected to VN1, VN3 and VN4 because in those

columns of H, representing the VN nodes, there are 1's, while in the first column, for example (VN0), there's a 0 and so CN0 is not connected to VN0.
Finally, the CNs have a different number of VNs connected to its (CN0 to three VNs, CN1 to four VNs, CN2 to two VNs, …) so this code is not a regular one.

What it's of most importance for the project aim, it's the LDPC decoding operation, by which, from a coded message, the original message transmitted must be extracted. There are many possible executable algorithms for that purpose, but the better onewhich is the considered on this work, is the *Message Passing Algorithm*. This process is based upon the continuos exchange of messages between the two sets of nodes connected to each others; the nodes are seen as computational points able to manipulate, and then send, data received in the previous phase, stored in appropriated cells of memories. The computational and sending operation phase of a group of node must be strictly separated from the other set one.

The VN nodes contain the original information, the LLR (Log-Likelihood Ratio) input as a logarithmic representation of the encoded bit. Every time space, in its computational phase, each Vn_j node, generate a message Q_{ji} for every CN_i connected to and a unique output to be stored in the memory, using all the messages previously received from the CNs:

$$output_j^{[k]} = \lambda_j + \Sigma R_{lj}^{[k-1]}$$

$$Q_{ji}[k] = \lambda_j + \Sigma\left(R_{lj}^{[k-1]}\right) - R_{ij}^{[k-1]}$$

where λ_j is the LLR input of the Vn_j node and *l* represent the index of all the CN nodes connected to the node VN_j. Here's indicated the *k* iteration and *k-1* represent the previous one.
So the output data for a node VN_j is calculated as the sum of its λ_j input and of all the R_{ij} messages previously received from all the CN_i nodes connected to its.
On the other hand, the Q_{ji} message for one CN_i is the sum of the λ_j input plus all the R_{ij} messages, except the one received from the CN_i to which the message is directed.

During the following stage of the process, every CN_i node will have been received all the $Q_{\beta i}$ messages from the VNs; when the computational step will begin it'll elaborate the R_{ij} next messages to deliver to each VN_j connected to it, performed as follows

$$R_{ij}^{[k]} = \psi^{-1}[P_i - \psi(Q_{ji}^{[k-1]})] \qquad where \qquad Pi = \Sigma\left(\psi(Q_{\beta i}^{[k-1]})\right)$$

As well as the VN node one, the CN message depends on the sum of all the previously received messages except for the one come from the VN node the message is directed too. But still, now a non linear function is involved, that makes all more difficult. $\psi(x)$ is a non-linear, non-limited function defined as:

$$\psi(x) = -\ln(\tanh\left|\frac{x}{2}\right|) = \ln\frac{\left(1+e^{-|x|}\right)}{\left(1-e^{-|x|}\right)}$$

Working with this function carries a lot of mathematical problems and makes the processor much more complex and heavy. Since the complexity of this function, a lot of approximations have been developed. For this implementation, the minsum approximation will be used, that simply consider the least one among all the $Q_{\beta i}$ messages received, except the one Q_{ji} previously received from the VN_j to which the message is directed:

$$R_{ij}^{[k]} = min_{\beta/j} |Q_{\beta i}^{[k-1]}|$$

That's clearly the simpliest approssimation might be used.

Summarily, the LDPC algorithm here implemented is composed by two different and sequential phases and it's for that reason called *Two Phases Message Passing* (TPMP) algorithm too:

1. firstly, all the VN nodes send the first message, composed only by the LLR inputs, to all the CNs connected to them;
2. the CN nodes, having received the Q messages from the VN nodes, elaborate the data and create an R message for each VN connected to them, then send it to all the VN nodes;
3. the VN nodes, having received the R messages from the CN nodes, combine the data with the input associated to each one creating the output, that will be stored in a dedicated memory, and new Q messages to send to all the CN nodes;
4. the process follows at point 2.

All the enumerated phases must be fully separated without overlapping, to ensure the correct execution of the algorithm.
This exchanging mechanism continues till the convergency of the algorithm and so the correction of the data input. Sometimes the convergency is puntual, so after a number more or less big of iterations, the result will arrive at a unique solution; other times the convergency is asymptotic, so the result oscillates between two values. In this case is sufficient to let the algorithm execute for a large time and then to take the obteined result.

At the end of the iterations, the Message Passing performs a large amount of data output, since the precision used for the ecoded bit is a 6-bits data information cell for each VN, i.e. 2304 bytes for WiMAX case. In fact, the encoded bit contained in the VN node is represented as the sign bit (the most significative one) in a 6-bit data information format.
Finally, when all the obteined 2304 bits are taken out, it's mandatory to remember that the nature of the LDPC code is a (2304, 1152) one: from the encoded 2304 bits will be obtain 1152 bits of decoded information, by multipling them with the H matrix:

$$\boldsymbol{x = cH}$$

it's easy to demontrate this relationship. Therefore, the encoded message came from multipling the x message with the generator matrix G ($c = xG$), so to come back to x from the code c it's necessary to reverse the G matrix,

$$G = [I \mid P] \quad =>^{(-1)} \quad [P^T \mid I] = H$$

and the reverse of G is the H parity-check matrix.

3.3 LDPC Decoder Conceptual Architecture

3.3.1 Architecture needs

Once studied the Message Passing algorithm that exploits the Tanner graph, the approach that should be applied to solving the problem must be specified. In the case of a hardware approach to the graph, then by using the physical creation of its nodes and interconnections between them, the number of resources would be very substantial, given the size of the matrix H. The project was thought by a software approach, in which the nodes are not physically built into the architecture.

What is needed in the architecture of the processor ASIP thus depends on the particular variety of tasks performed. It is seen as the Message Passing algorithm performs calculations internally to nodes, such as sending and receiving packets of data, creation of output and processing input data.

Regarding the management of data packets, this function is performed entirely by the component PHAL, which should be composed of multiple functional blocks, because it also has other tasks: the package management control and synchronization, mapping of the network, ...

The ASIP, however, will be responsible for receiving and storing messages contained in the packages into dedicated memory. Since the approach focuses on flexibility, as mentioned above, the nodes contained inside the ASIP can be anyone, thus they're not tied to a single array H. The ASIP will receive and send packets with data and indications of the source node and destination node, which will interpret to deposit it in different memories depending on the buffer fields and algorithm step.
Will then be needed, apart from a program memory in which to store the main program that performs the Message Passing algorithm, other memories as separate. Besides this, of course, the architecture of the parallel type is provided, then served by a system of pipelines. Several buffers will be used to temporarily contain the input or output or for purpose of calculation. For the messages calculation will be necessary to use an arithmetic logic structure, in this context having the form of a DSP (Digital Signal Processing) located inside the platform.

LDPC algorithm functionalities	**Processor Involved**	**Architecture Resource**
Control and synchronization packages management	PHAL	FIFOs, ALU
Network mapping	PHAL	FIFOs, ALU
Message packages forwarding	PHAL	FIFOs, ALU
Creation messages by data messages and data input elaboration	ASIP	Memories, buses, temporary registers, DSP
Creation output	ASIP	Memories, buses, temporary registers, DSP
Algorithm code flow	ASIP	Program Memory, program memory bus, pipeline, customized instruction set
FIFO-based communications	ASIP, PHAL	FIFO-based interfaces

Therefore, in summary, the feature set provided by each processor element (PE) of the network, divided for PHAL and ASIP, as the LDPC decoding algorithm has been thought, is composed by all the components shown in the table above.

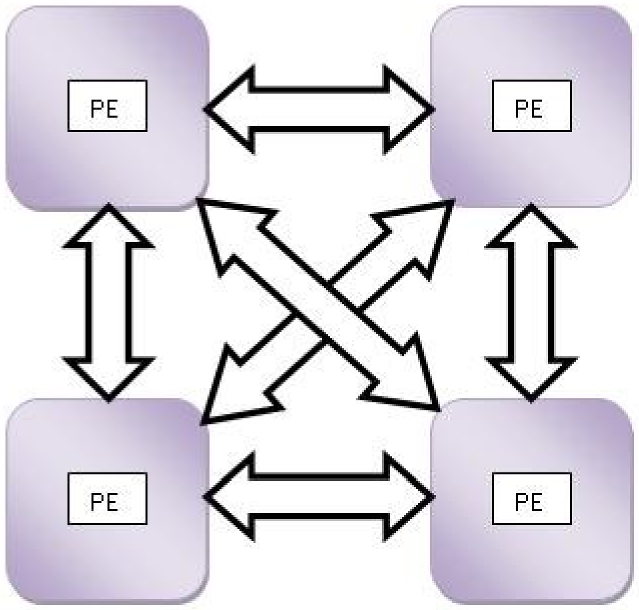
The PHAL component only needs of several FIFO where to store the incoming packets and an ALU system, because it has only to check the fields of the buffer to interpret, thus to forward it through the correct destination. Was chosen to use a standard FIFO-based communication system made of data buses and control signal for comfort of use: its operation will be explained later.

The ASIP has much more duties and responsabilities into the LDPC decodification, since it must keep memory of the incoming messages to then process them and create new messages and output. So it will need several memories where to store data messages and the origin/destination nodes information and DSPs to perform calculations. Finally, since the Message Passing algorithm is exploited through an assembler instruction set code, must be given an extra memory where to save the code and a set of instructions that performs the algorithm.

3.3.2 Proposed Architecture based on NoCs

The Message Passing algorithm operations, the LDPC decoder requires, are very heavy to support. This depends on the fact that a large amount of data must be elaborated: the H matrix of the WiMAX case is *1152x2304*, so the Tanner Graph system needs 2304 VN nodes and 1152 CN nodes. By using a single processor element to perform the algorithm, all the nodes should be allocated in only one component. This approach would occupy a great bandwidth and a lot of resources, with the consequence to be very slow and to consume a lot of energy.
The idea is to split the nodes among more processors ASIP, as reduce the complexity and the power consumption (joined to the time of execution) of each one. There will be no longer a unique Processor Element (PE) but different PEs connected in a dedicated network, or a Network on Chip (NoC). In advance, this would be a specific NoC, because the protocol of communication and the performed operations are specific for the LDPC decoding algorithm aim, becoming an ASNoC (Application Specific NoC).

The entire structure can be illustrated like an all-connected network of equal elements in the figure below.

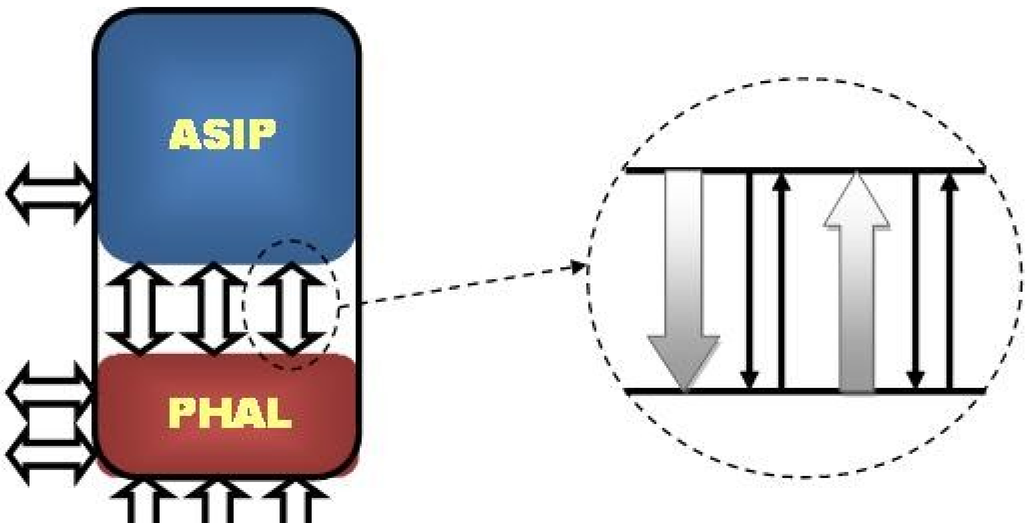


The network is composed by 4 PE interconnected each others by 2 pair of buses, including a 32-bit word through which the data will be delivered plus some singol bit control signals. All the components have a unique clock, distributed by an external Linux master controller that delivers its control message packets via synchronization and control interfaces.
Every PE is formed by the union of 2 separeted components, as already said: one ASIP, that contains a part of VN nodes and a part of CN nodes and it's responsible of the calculation tasks and output storing; one PHAL, a processor working as a router, whose duty is to deliver messages and to manage packets through the network.

3.3.2.1 The NoC interfaces

The PHAL is the only part of the PE responsible to communicate with the ASIP and with all the others processor PHALs; a simple scheme is shown in figure below.
Every ASIP has 3 interfaces with its PHAL, one for each PE to which the message can be delivered; in this manner, the PHAL can be built with an easier logic, since it automatical knows that if a word came from a virtual interface (the connection ASIP-PHAL), it must be sent to a particular output interface (the conection PHAL-PHAL). Hence the PHAL needs only a simple routing map associating one virtual interface with one and only one data output interface.

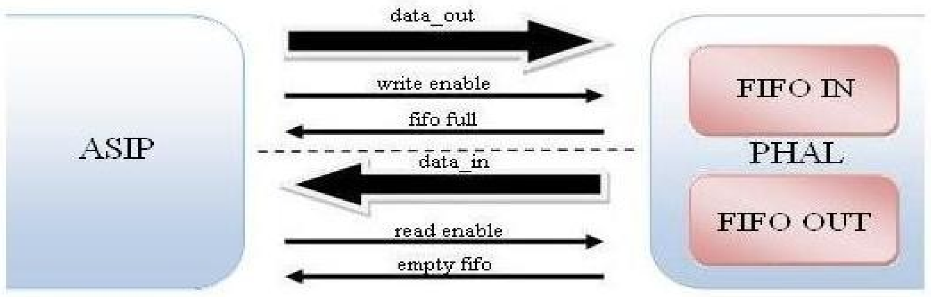
There are different type of interfaces; indeed, there's an interface ASIP-PHAL, one ASIP-Linux, a PHAL-PHAL interface and a PHAL-Linux one.
Every interface as a behaviour different from the others but the common thing is that each one is made of two 32-bits word bus data, one for the input and one for the output, plus four control signal, two for each data bus (one empty-fifo controller and one read-enable for the output plus a fifo-full controller and a write-enable for the input).



The distinction between the interfaces depends on how these signals affect the logic of each component.
Before to explain the way of working of the different interfaces is important to describe the storing method. The ASIP interfaces aren't fifo register systems; they don't have temporary registers where to store information packets to deliver, because they process the arriving data and they need it for a longer time. Because of that, they have several memories where to store data bytes. Instead, every

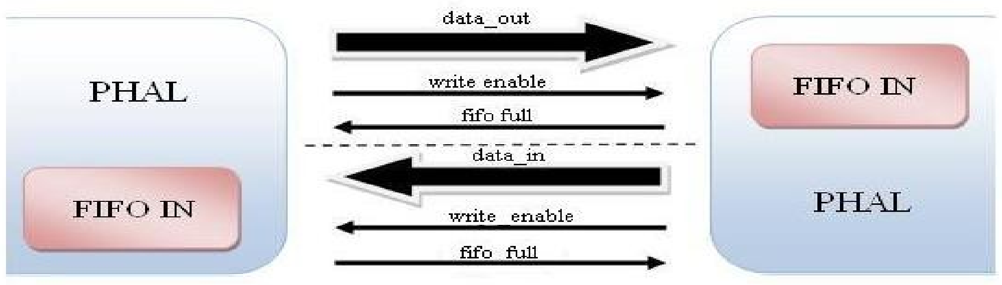
PHAL has a complex fifo interface system: two fifos (one for input and one for output) for data packets to and from each ASIP interface; one input fifo for the data arriving from other PEs; two fifos for the control and two fifos for the synchronization interfaces.
Let's see all:

- **ASIP-PHAL** : the ASIP doesn't have fifos, so it has to store the input data directly in memory; similarly, when it has data to send, it must throw it out immediately. So, the PHAL must have two fifos where to store data temporarily.



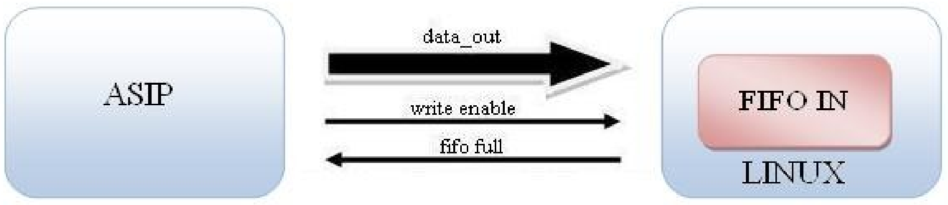
If the ASIP needs to send data, looking the fifo full control signal from the PHAL output fifo for that it doesn't get filled, it set the write enable and put the word in the output 32-bit bus. Meanwhile, when the PHAL input fifo contains some data word for the ASIP, it puts low the empty fifo control signal; in that manner, the ASIP will know that there's some data to adquire. By settin the read enable control signal it will get the word to be stored in its memory.

- **PHAL-PHAL** : two PEs exchange data word informations through its PHAL. Since the operationws are managed by only one FSM each component, when data's arriving it might be performing other tasks. For that, it's fitted with an input fifo, so that the output data can be throw out anytime it's desired: although the PHAL is doing other tasks and it can't look at the input interfaces, incoming data is stored in the fifo registers and the PHAL will check the packet later, when it'll have time to do it.



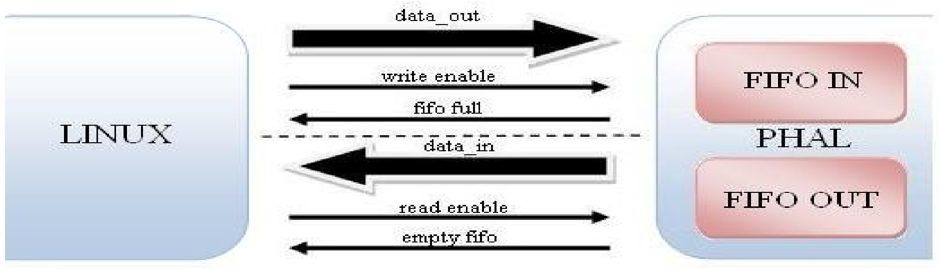
The signals are the same but are managed in a different way. The data sending out is performed by each PHAL as the ASIP does: it throws out words through the 32-bit data out bus and setting the write enable, checking the fifo doesn't get filled. But for the reading phase, the PHAL doesn't look at the control input signals. In fact, the PHALinput interface has an incorporated fifo working alone, storing data input in its registers. The PHAL looks directly at its empty fifo signal to obtein incoming packets.

- **ASIP-Linux** : between each ASIP and the Linux Controller, there's a particular communication chanel (*virtual interface 0*) dedicated to the exchange of initialization and final information data packets. In this case, in which the initialization is performed statically, this interface is used only by the ASIP sending the output data words to the Linux Controller at the end of the LDPC decoding iterations.



When the ASIP arrives at the last operation step, delivering out the algorithm results, it sends to the Linux Controller the data in as well as all the other interfaces, through a 32-bit width bus and setting the write enable.

- **PHAL-Linux** : the aim of the PHAL is primarly to monitor the correct execution of every process task. It has to receive orders from the master and report the status of the algorithm, so it needs two interfaces to the Linux Controller, one to and one from that.
In advance, the synchronization Deamon in the PHAL needs a periodical communication with its master, to request the correct time and synchronize it to one unique time clock; so it needs two more interfaces to the Linux Controller.



The Linux master controller has to face with many duties and tasks. It will not be always able to check incoming pakcets form the PEs. Therefore, the ideal way of exchanging information is that the master could decide each time when do adquire the informations contained in the PHAL fifos by controlling the fifo full signals. Each PHAL will store data in an output fifo connected to the master and input data information form the master will be stored in an input fifo internal the PHAL component. In total, the PHAL, with two register fifos for control operations and two for synchronization operations, will be able to manage all the communications with the master control.
The Linux Controller is directly controlling the exchanging performance with the PHAL, while this is only concern of watching constantly at the fifo input and charging the fifo output with data to throw out.

3.3.2.2 The Master Control

Theoretically, one of the four PE should act as the master, ie set a master clock time to synchronize all the other elements and manage the execution orders of the LDPC algorithm. In practice, as the NoC has been implemented with all-similar PEs, neither of the processing elements has the power to influence the instruction flow. The role of arbiter is played by an external Linux-based controller inglobing all the network and acting for the components from an higher point of view.

Communicating rather with the PHAL then the ASIP, it requires the commands to manage the execution by delivering some information packets to each PHAL. So the Linux external controller take in charge when loading the applications onto the ASIPs, when beginning the initialization procedure and running up. Eventually, as the figure of a user, it makes the LDPC algorithm converging, establishing how much time slot cycles are necessary for the error correction system.

The only real function dedicated to the PHAL is the time slot division, being up to this component setting the ENABLE time slot signal to the ASIP. Actually, the time slot duration has been previously decided by the PHAL code programmer.

There might be a future improving implementation. One another possibility, in fact, could be that the Linux (like any other) master controller, during the initialization phase send to each PHAL the time slot duration required by the LDCP complexity. Other still better solution could be to make it *flexible*, i.e. let the PHAL change the time slot duration value. With an additional part, the PHAL could check if all messages are sent in time before the beginning of the next time slot, and so to lengthen or to shorten this time dinamically.

It'll be explained, in later chapters, as the external control relies on VPI libraries to function and to communicate with the various elements of the network.

4 PHAL Concepts

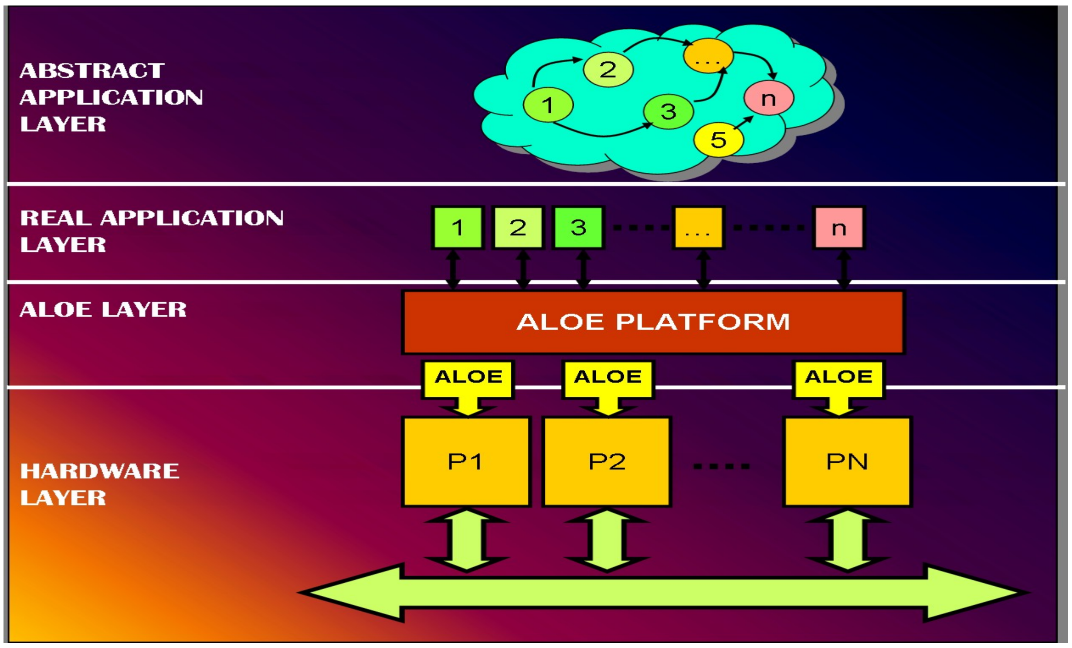
The PHAL concept tries to address the main issues related with the Software Radio technology. It assumes the need of supporting reconfiguration of the radio processing chain, the need to deploy the radio application onto a heterogeneous hardware platform capable to assume the high computing requirements of modern wireless systems, the need to assure strong real-time execution constraints, the need to easily develop and integrate software modules and the possibility to execute them in different types of processors assuming an easy portability.

One of the most relevant objectives in the process of defining a common framework to develop and deploy software radio applications is to eliminate platform (hardware and support software) dependencies. On the other hand, radio applications are built through a set of software modules (hardware or software based) that communicates among them. The common used terms denominates such modules as “objects”. Therefore each one of such objects are signal processing blocks, some of them with important requirements in terms of computing resources, that needs to acquire/deliver information from/to other objects through proper interfaces. Nevertheless, one of the most relevant assumptions of PHAL-OE is that such interfaces between objects are unknown at object development time. Only at execution time all the required objects will be integrated and the entire radio application will be built.

All these objectives have pushed us to define and develop a multi-platform software abstraction layer and execution environment, ALOE (Abstraction Layer & Operating Environment), capable to provide such features. The list of the main functionalities where the PHAL-OE must provide support on includes:

- **Flexibility :** the framework must efficiently implement the flexibility concept required by SDR. It is assumed to be based in the capacity to facilitate the reconfiguration as the basic mechanism providing flexibility;
- **Execution control management :** the coordinated execution of the whole system must be assured;
- **Hide the platform heterogeneity to the radio application :** abstraction layers are required are basic mechanisms to provide such feature;
- **Computing resource management :** under a scenario with limited computing resources the need of a specific mechanism to manage the available resources promotes the efficiency in the whole system and extends its interactions with the radio resources management part. In addition it is capable to assure the overcoming of the real-time constraints;
- **Data packet oriented messaging:** not processor (device) oriented communication mechanisms. Data packet oriented communication network among the heterogeneous processors is built;
- **Parameter or variable (signals) evolution capture** during the execution of the application;
- **Processing resource parameters evolution captured** for autonomous management or control;
- **Auto-learning/cognitive capabilities** for the internal resource management can be easily incorporated;
- **Support to the Cognitive Radio strategies** thanks to the capacity to capture, in a

coordinated way, relevant information from different layers of the radio and computing system;



Previouse figure tries to illustrate the PHAL-OE concept. On the bottom we can see the Hardware Layer where we can found several processors or Processing Elements (PE) physically interconnected among them. On top of the figure we can see the Abstract Application Layer where a graph of a radio application tries to define the required tasks (objects) and the data flow among them. In the Real Application Layer we can see the previous tasks but using the services or functionalities provided by PHAL-OE to build coherently the radio application. On the PHAL-OE Layer we observe that from the point of view of the tasks (objects) conforming the radio application all them only see the same platform, the PHAL-OE Platform, not being aware of any detail related with the hardware. On such layer, but from the hardware point of view, we can find the specific implementation of the PHAL-OE functions for each one of the processors used to build the entire Hardware Layer.

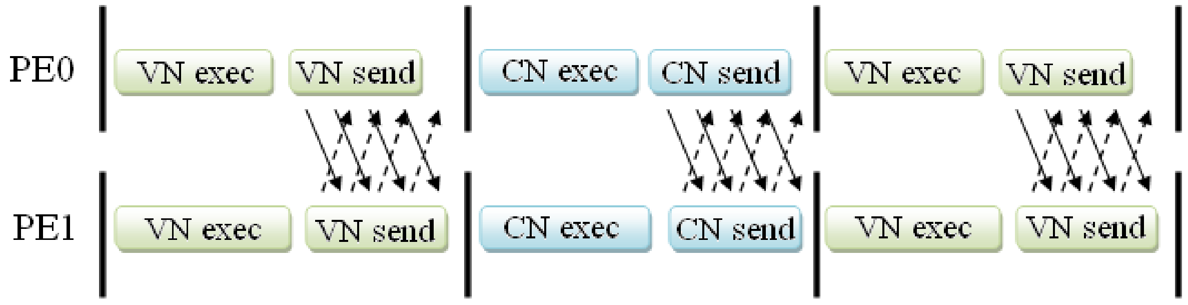
5 Implementation

5.1 The PHAL Component

Let's focus on this section of the whole PE. The PHAL component is the peripheral part of the Processing Element or it can be better considered as the interface between the ASIP (ie the real processing part) and the external world. Indeed, it's basically responsible of the routing operations, since it controls the packet traffic to and from the ASIP component, and so permitting the data exchanging through the entire network.
Plus, the PHAL has the duty to manage the control and synchronization packets coming from a master that's been controlling the LDPC decoding operation: if the entire system was a network made of several elements, it would have to control and synchronize it.

5.1.1 The Time Slot Management

As it's been said, each ASIP in the PE must be able to execute the VN node operations and the CN operations alternatively. Since the two operations must be performed separately, it's mandatory to implement a time slot, in which each operation node phase is alternately executed without overlapping: for this reason, the PHAL has a timer inside that scan the enable execution time of each operation.



Each time slot, the ASIP, alternately, performs the node operation (Variable or Check) as a calculation phase and then sends all the messages created through its data interfaces. The associated PHAL gets these messages and delivers it into the network.

The time slot must be calculated for there will be time enough to perform the node calculation and sending operations. In fact, at the beginning of the next time slot and so of another calculation step, all the data must already be arrived to destination and stored in memory.
The PHAL, following the master directive, worries that its ASIP does its job and does it in a resonable time; it controls the change of status of the ASIP and the time partitioning of all the system, including the time slots.

5.1.2 The PHAL Architecture

Unlike the ASIP, that's a SystemC code based processor, the PHAL has been created using the VHDL hardware code (Vhsic Hardware Description Language). That's because this part wasn't thought to be too large, being only a router component and the idea of using an ASIP-like microprocessor, with dedicated resources and instructions, seemed a waste of space.
On the contrary, a VHDL-based architecture allows to have a FSM with very few resources and the possibility to join different pieces; finally, all the component is implementable on a FPGA board with the characteristics shown in the table below:

Number of Registers	9193
Number of Look Up Table	6561
Number of IOs	583
Minimum Clock Period	4.828 ns
Maximum Frequency	207.108 MHz

It's possible to see in the figure in the following pages that the whole PHAL system is formed by the union of different blocks joined together to perform all the basic communication and control tasks. The fundamental idea is the modularity feature, hence the possibility to add or remove blocks, adding operations (such as the statistical one) or removing its from the scenario. There's no danger of signal conflicts because each input/output and each register is managed by only one block at a time.

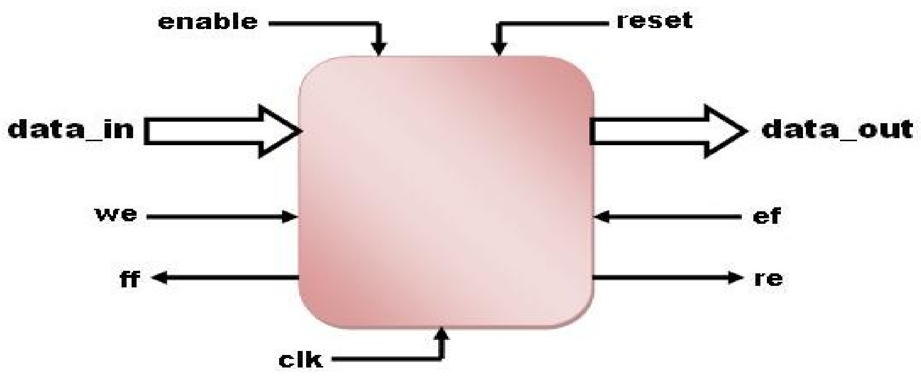
A block unit is thought to act as a Deamon, performing a specific task and aim, capable to control only a part of the input/output signals and of the resources, but responsible of an entire process. They're composed, each one in a different way, by a finite state machine (FSM) having the role of signal managing, data processing and output delivering. The interface mechanism of each Deamon block is fifo-based: all the signals moving through the network from or to other Processing Elements are stored in fifo registers controlled by the FSM in each block through the signals in the fifos. The number of fifos and kind of interchanging between blocks will be discussed in an other chapter.

5.1.3 The FIFO Component

The fifos used in the PHAL component belong to the same model, with the same mode of function. The main shape of this component is shown in a figure of the next page.
The considered component is more then a fifo, because it manages a real complex storing function. It's composed mainly by a parallel-in-parallel-out collection of registers, in the format of 32-bits words. There's only one clock signal entering, so that it's completely synchronous, with the same temporization for the Writing process and for the Reading process. It has an asynchronous reset input, connected to the outsider general reset that clean all the registers and set its to an all-zeros value. Plus, even if not essential, it's performed an enable input that actives the fifo when pulled at a high level.
For the operations of writing and reading, the system manages the parallel data input and the parallel data output (of a word dimension) with a write enable and a read enable and two others wire reporting the state of the fifo (full in one case and empty in the other one).

When the write enable is pulled up, the fifo block gets the word that finds at the *data_in* input and stores it into the first free register, once every clock period (on the rising edge) until the write enable (*we*) is at logic ‘1’; for sure, when in the fifo there will be no free space, the fifo full output (*ff*) will be internally pushed to high level. For that reason, the external agent pushing data into the component must control that signal and stops in case of fifo full to prevent the overwrite of informations already stored in the register memory.

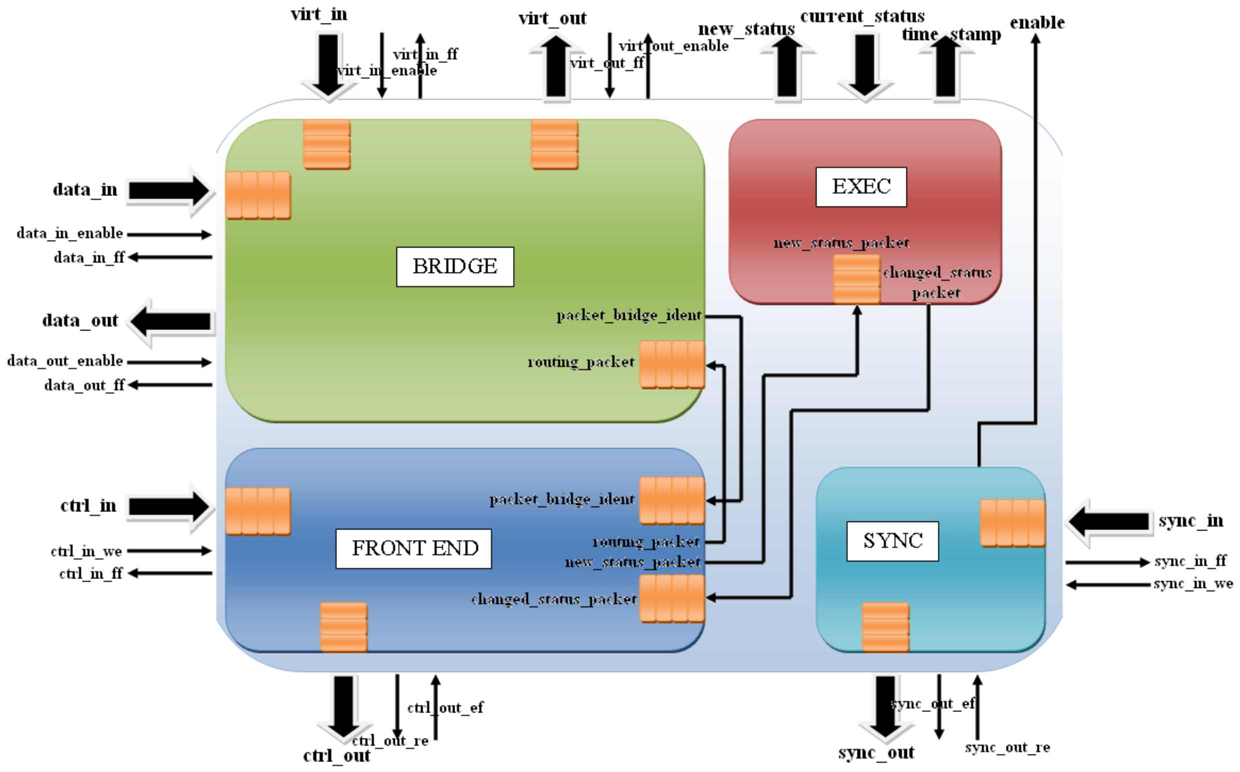
On the other side, the fifo is always pushing out on the *data_out* output pin the first word in the registers, but till the read enable (*re*) will not be pulled up to ‘1’, it will not throw out anything.



Generally, monitoring the empty fifo signal (*ef*), the outside agent can realice when information data is stored, put the read enable input to ‘1’ and start catching words, one for each clock period. Every time that the read enable is put to ‘1’ the data is thrown out and borrowed from the internal registers. When, finally, the fifo will not have more words, the empty fifo will be put to '0'.

5.1.4 The Deamon Components

Let's take a look now at the higher level, the PHAL.



The actual version is made of 4 main blocks:

- FRONT END: it's responsible of the command and control operations of the whole system. It receives the command packets from the master controller, that are stored in a dedicated fifo. Moreover, the outgoing packets are not sent directly to the master, but they're stored in another fifo so that, when the master could, it will catch it in order to obtain the PHAL informations.
  It has two other fifos, by which it can receive informations from other blocks of the PHAL and so better manage the correct execution of the algorithm. One of these fifo receives the informations on the changing status process from the EXEC block. Each time the status of the ASIP has been changed by the EXEC Deamon, this has to make known the FRONT END of the new status. It does it by sending a packet to this fifo.
  The other fifo is done to contain the packet received by the BRIDGE block during the bridge identification operation.
  There's only one FSM controlling the execution of all the commands in the entire Deamon. After an initial reset, to make the machine began in a known status, the "Registration to the master" packet is built. It carryes the ID randomly proposed, so the fifo *ctrl_out* is filled with all the packet, waiting for the master removing all.
  Then, the FSM remains in a waiting state, monitoring the empty fifo signal of each fifo from where new packets could come: the *ctrl_in*, the *packet_bridge_ident* or the *changed_status_packet* one. And every time one of this contains a packet information, the FSM flow follows a different operation branch.
  When the *packet_bridge_ident* fifo is no longer empty, it means that the bridge identification operation must be performed. The FSM takes all that packet, put a header and send it by the *ctrl_out* interface to the master.
  In the other case, when the EXEC block responds to a changing status process and returns the actual status of the ASIP by the *changed_status_packet* fifo, the FRONT END communicate it to the master, by building the "Watch object status change" packet and sending it through the *ctrl_out* interface.
  When's arriving a packet from the *ctrl_in* interface, the PHAL firstly realizes two control packet operations: checking the first word (must be the HEADER MAGIC constant value, otherwise the packet is trashed) then controlling the size of the received packet having the same of the one reported in the header. Finally it looks at the type of command and executes:

  1. FRONTEND_REGISTER_ACK : it's the response of the master to the request of an ID made at the boot. The FRONT END changes the ID and communicates it to the BRIDGE;

  2. EXEC_SETSTATUS : it's the command to change the ASIP status. The information of the new status is sent by the *new_status_packet* output by filling the fifo in the EXEC block;

  3. BRIDGE_ADDITF : this packet containes the field to create the routing table of input (external to virtual interface) and output (virtual to external interface) in the BRIDGE component. The four words in the packet with the routing informations (input/output, header, virt_itf and ext_itf) are then sent to the fifo in the BRIDGE block;

  4. BRIDGE_IDENT : when receiving this command, the FRONT END advertises the bridge that must deliver a packet through each *data_out*, to make known all its neighbours of its presence. And to avoid the waste of another dedicated fifo, a single

word packet is sent to the routing fifo of the bridge, that'll be aware of this command because of its format.

- SYNC : the synchronization component is made of two fifos, connected to the interfaces *sync_in* and *sync_out* allowing the communication with the master, and one timer counter. This is the only block controlling the temporization and the time slots. In fact, synchronous with the clock, it punctuates the time and returns to the system the count of microseconds and seconds. But its main task involves the indication of a new time slot beginning to the ASIP via the ENABLE pin activation.
  Because of inaccuracies, the timer watch is subject to drift; to avoid this problem, the SYNC block periodically sends to the master a synchronization request, to which the master will responde with its master time. This will be used to correct each SYNC Deamon time.
  The FSM of the SYNC block provides, like the one of the FRONT END, a boot stage. During this first step, it creates the "Register to sync_master" packet and send it through the *sync_out* interface. Then it just lays in a waiting step, while it keeps monitoring if something arrives through the *sync_in* interface. In this case, when a packet arrives from the master control, it's the "Synchronization Acknowledgment Packet", containing the master time to correct the SYNC timer. Otherwise, periodically, the SYNC creates the "Synchronization Request" packet with the own timer time and sends it through the *sync_out* interface to the master.

- EXEC : the EXEC block manages the changing status operations of the ASIP. It's furnished with only one fifo to receive the packet from the FRONTEND, with the order for the new status to the ASIP. The FSM is always waiting for a new status notification arriving into the fifo and when it happens, it collects the basic informations (object, status and time stamp) and transmits the new status to the ASIP. Then, it waits the ASIP updating its current status and communicates it to the EXEC. When the EXEC will receive the current status of the ASIP LDPC algorithm, it will communicate it to the FRONT END block, this last in charge of submit the status to the external master.

- BRIDGE : it's in charge of deliver the packets arriving from the external interfaces to the correct virtual interface and vice versa. It has one fifo for each virtual input interface (data arriving from the ASIP to send outside) and one for each external input interface (data arriving from outside to deliver to the ASIP), plus one by which the informations on how to build the routing tables from the FRONT END arrives.
  It does nothing but to keep controlling the empty fifo signal of all the input data fifos, checking if there's a packet arriving:

  1. If something arrives into the routing parameters fifo, it could be the BRIDGE IDENTIFICATION COMMAND or the parameters to fill a routing table. In the first case, it throw out through all the *data_out* interface a packet for the identification process; if it's for a routing table, it fills the right table (input or output is indicated in the arrived packet) with the informations of destination and header;

2. If there’s a data input fifo not empty, it goes catching the packet, it looks at the header, searches in the routing table input the corresponding virtual interface and forward through it the information;

3. If there’s a virtual input fifo not empty, it goes catching the packet, it searches in the routing table output the header which corresponds to that virtual interface and deliver outside all through the data out interface indicated.

PHAL tasks are completely indipendent from the ASIP purposes. As a matter of fact, the ASIP depends on PHAL performance and operation managing. This makes the PHAL the most important device into a NoC. Furthermore, the ASIP task, as implementing the message passing algorithm, is partitioned into time slots and it's not able to begin the execution phase without an ENABLE activation, performed by the PHAL component. On the other hand, PHAL is always executing and checking the network functionality on one side, while scanning the ASIP operations and managing its packet delivering on the other.

5.2 LDPC Components: VN, CN

ASIP is the acronym of Advanced-Special Instruction Processor, a customized microprocessor designed to perform only a specific kind of instruction set internally to the program main code. Its architecture and instructions are built for a specific purpose, not for general use.
The software instrument by which the processor's been developed is the LisaTek Coware programmer device. This framework provides the possibility to declare all the needed resources, as registers, memories, pipelines and so on, plus to implement and describe the instructions that permit to realize the desired aim. By this tool it's possible to declare the instruction set to better perform the LDPC algorithm minimizing the resource allocation.

5.2.1 The ASIP Architecture

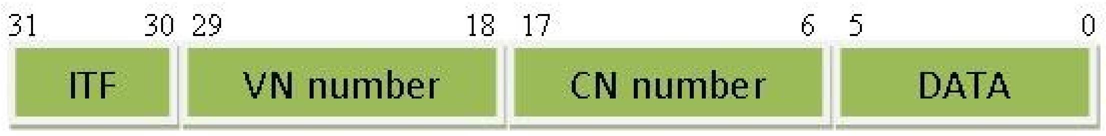
The CoWare working platform is composed by several fields, each one with different features; the principal one is the *main*, reported in the Appendix A. Into this area, takes room the resources declaration, the main and reset part of execution code, the different groups of instructions and the instructions automatically executed during the fetch part of the decoding operation.

5.2.1.1 The memory set

Let's begin seeing at the resources declaration of the memory set.
It's mandatory to specify every memory, comprised the program code one. Joint to the memory, the respective bus must be allocated, one for memory, to carry data words during either reading and writing operations. The LDPC decoding ASIP has a set of memories with different purposes; each one of them it's thought to host a specifical kind of data and data bit format. The great amount of memories (normally should be one for the program code and one for the data) will require a lot of heavy logic to be managed. Though it might be better organized in a unique whole memory, to reduce the access logic, it's now necessary because of the access to memory limited to one acces for cycle period.

Substantially there’s a **program memory**, where the program code executed for the algorithm purpose is stored and others six memories containing different nature data:

- **VNi_mem** : it contains the connections between each VN node and the CN nodes, with the data processed by the VN to be sent to the relative CN.



Each bus number addresses a block of 32 bits, divided into several fields.
Each connection, in a word, is represented by indicating the pair of nodes (the VN one and the CN one) joined together and the information data.

The interface field indicates through which virtual interface this word must be sent. It requires only 2 bits because the NoC provides 4 PEs and each interface is directed to one of the other three PEs. Besides, the interface “00” doesn't indicate an external PE destination, but the same of the origin. In this case the target lays in the same PE of origin and it won’t be delivered outside but directly stored in memory.

The VN node field indicates the number by which is performed the identification of each VN in the entire network.

The CN node field shows the CN node joined to the VN.

The data field is occupied by the Q message, elaborated by the VN in the word and directed to that specific CN. In the beginning, it will be filled with the input LLR data, the same data message to all the CNs connected to the same VN.

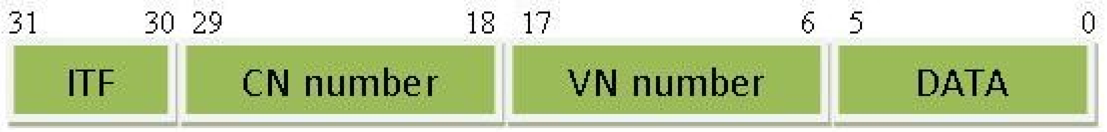
There’s a dedicated interface (*virtual data 0*), connecting the ASIP and the Linux controller, by which it’ll be possible, in a future expansion, to charge this memory (and the CNi_mem too), during the initialization stage, with the node connections expressed in the Tanner Graph. Right now, the data connection (with input LLR in the VNi_mem) is stored in the static way, by evocation in the main program code.

The format choice is due to the 32-bit word standard of use. The VN and CN nodes, in the WiMAX standard implementation, are expressed by identification numbers representable in a 12-bit format. Indicating the origin node and the destination one occupies 24 bit. Other 6 bits are mandatory to carry the message data. 2 bits lack to fill the entire buffer and they're dedicated to the interface number.

Surely, if the NoC were composed by more then four PE or the Tanner Graph were made of a greater number of nodes, no longer expressable on 12 bits, a single 32-bits buffer wouldn't be enough and a longer one would be necessary. Or another field buffer organization should be performed. Neverthless, this project is customized to the WiMAX requirements, not to other ones. The ASIPs don't know which nodes they contein, so in every data packet is necessary to indicate origin and destination. Furthermore, only the PHAL knows how to deliver packets in the network but it doesn't own the indication of every node's position: it only knows by which interface they came from and the interface field helps the delivering task.

- **CNi_mem** : it’s basically organized as the VNi_mem, except for the field order (it contains the connections between each CN node and the VN nodes with the R messages).

Each bus number addresses a block of 32 bits, divided into several fields,



The data field is filled by the R message produced by CN to the VN. In the beginning, this field is empty (there are zero bits) because no messages were exchanged yet; in fact, the first operation is the VN input sending and only then the CN nodes will elaborate and replace the message.

For this memory too, the connection storage will be made statically.

- **VN_to_CN_mem :** it has the same format of the VNi_mem, because it contains the Q messages received by the VNs from other PEs and directed to the CNs contained in the PE of this memory.
  The difference with the VNi_mem is that, into this memory, messages are ordered by increasing number of CN. That's because of comfort: when, during the creating R message operations by the CN nodes, it will be necessary to obtein the Q messages in the VN_to_CN_mem, the LDPC algorithm in the ASIP will know exactly where to find each message related to a known CN.

- **CN_to_VN_mem :** it has the same format of the CNi_mem, because it contains the R messages received by the CNs from other PEs and directed to the VNs contained in the PE of this memory.
  The difference is that, into this memory, these messages are ordered by increasing number of VN. That's because of comfort: when, during the creating Q message operations by the VN nodes, it will be necessary to obtein the R messages in the CN_to_VN_mem, the LDPC algorithm in the ASIP will know exactly where to find each message related to a known VN.

- **Input memory :** it was thought as a 8-bit width RAM because of the standard memory size based on multiples of 8 bit cells. Neverthless, only 6 bit will be important, since it must contain the input LLR 6-bits data of the VN nodes. Actually, in the beginning, the data is already contained in each connection word of the VNi_mem memory. The reason why it's been decided to dedicate an entire memory to these data is that every VN calculation phase, the VN node needs its LLR to perform the next step messages and they can't be borrowed by newest incoming messages. Therefore, in the initialization step, the ASIP gets the input data (ones each VN) and stores it in this memory.

- **Output memory :** created as 8-bit width RAM for the same reason of the VNi_mem, conteins the refreshed output of each VN execution iteration.
  From those data will be delivered the sign bit to the control master through the virtual data 0 interface, at the end of all the requested iterations.

It's important to underline that, except for the input and output ones, in which each VN occupies only one byte cell, all the memories allocate more than one cell dedicated to each VN (in the VNi_mem and in the CN_to_VN_mem cases) or CN. That's mandatory, thinking that every memory describes the connections among VNs and CNs: each VN node might be connected to other 6 CN nodes and each CN node to 7 VN nodes, in the WiMAX implementation. But, since the

WiMAX standard implements an irregular LDPC decoding algorithm, those indicated are the maximum numbers of connections. And since it has never known how much node, each node is connected to, 6 word cells for each VN in the VNi_mem and in the CN_to_VN_mem and 7 word cells for each CN in the CNi_mem and in the VN_to_CN_mem must be reserved, although not all of them will be filled with a message.

On the other hand, this make the algorithm execution easier, because, since the maximum number of CN and VN in each PE is well known, the procedure can be, simply, scanning 6 or 7 words for VN or CN, knowing that they contain an all-zeros value if they're empty (there are no more connections, i.e. less than 6 or 7), in this way creating an automatic control cryteria.
For that reason too, in the beginning of the main program code, when the memories are filled with the words representing the connections, all the 6 reserved words for the VNs and the 7 reserved words for the CNs will be filled. If the nodes don't reach the maximum number of connections, the word will be filled with an all-zeros word.

Splitting the data memory in so many different blocks is due to the nature of the *Message Passing Algorithm*. When, as it will be shown, the code begins to deliver packet messages to other PEs, or even the same, these must be stored somewhere.
For example, during the Q messages sending operation, all the words in the VNi_mem, containing the connections of CNs with each VN, are sent. When received, they can't be stored in the VNi_mem because does exist the possibility to overwrite messages not sent yet. Similarly, it's forbidden to store them in the CNi_mem because the connection list of the CN nodes would get lost; or in anycase, it would be harder to manage the R messages calculation operation later.
Implementing other two memories, the VN_to_CN_mem for the Q messages forwarded by the VNs to the CNs and the CN_to_VN_mem for the R messages delivered by the CNs to the VNs, the iterations are more ordered and simpler.

5.2.1.2 The pipeline

It's been discussed how much the LDPC algorithm implemented into the WiMAX environment is heavy to support. And since the calculation procedures could last a long time, it's better to improve the processor performances as instruction branches management and normal instruction flow too (fetching-decoding-execution).
Buying time is possible realizing a pipelined architecture, ables to execute several tasks in one clock period. As every single part of the processor is personally designed, the pipeline stages declaration must be introduced too.

Still in the resources section, the pipeline declaration done by three stages: Fetch, Decode and Execute, plus all the Program Counters for each stage and the specific Program Counter for the branch operations.
Basically, the PC is the counter of the operations performed, while the Address Register (*AR*) keeps memory of the address where to point: that will be the address in the program code part of memory where the hardware will extract the instruction with which fill the Instruction Register (*IR*).

In the *fetch* stage of the pipeline, the IR is charged with the instruction to perform and the PC keeps on the counting operation; during the second stage, *decode*, the instruction contained in the IR is decoded and interpreted, while the registers needed for the operations are evocated; finally, the last stage *execute*, is responsible of the operation execution, by implementing the command and storing the results in the data memory.

The grace of a pipelined architecture is the chance to perform an instruction per clock cycle, in order to obtein a continous program code flowing. Indeed, a part three initial cycles needed by the pipe to be filled, every time an instruction is executed, the next one is already decoded and the following fetched. In other words, while an instruction is executed, concerning the calculation and

storing operations, it already begun the manipolation of the next addressed instruction and the next next one(i.e. the next two rows of program code instruction).
The one just described is the procedure followed if a branch doesn't occur. In that case, all the previously charged instructions must be thrown to charge the next one where the program jumped, through a stall pipe operation that avoid the instruction in the decode stage to be executed and a flush pipe operation performed in the fetch stage to clean the pipe. The only advertice is try to use as less jumps as possible to prevent the continued stale in the pipe that would slow down and render ineffective this type of architecture.

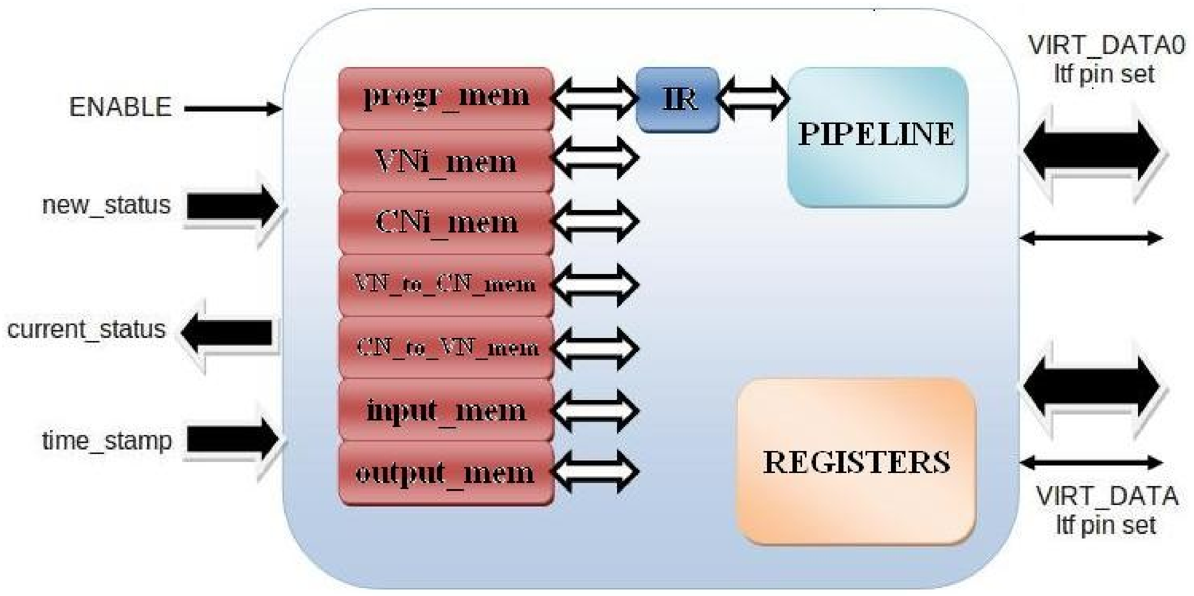
5.2.1.3 The whole block and the external interfaces

A part from the memory and pipeline declaration, all the needed registers plus the input/output pins for making possible interfacing with the external world must be declared.
The ASIP architecture can be designed as that below.

The whole block presents an input *Enable*, through which the PHAL informs the ASIP main program the beginning of a new time slot.

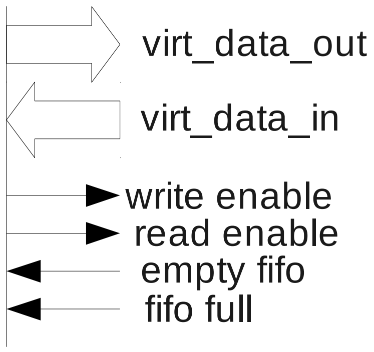
A 2-bits input, *New Status*, is charged into two 1-bit registers controlling the change of status and consequently the evolution of the machine, while the *Current Status* 2-bits output allows the PHAL to know every moment in which state the ASIP's executing its tasks.
The *Time Stamp,* as a hexadecimal value frame of 32-bits, provides the time indication of the current temporal slot. In our implementation this signal wasn't really used, but it could be used to delay the beginning of the calculations to a prefixed ordered time slot.

The group of virtual data pins represents the interconnections with the external world, performed by the PHAL and the Linux master control. They are only used for the data exchanging.
The first group, the *Virtual Data 0* interface, connects the ASIP with the Linux Controller bi-directionally: in one way, the Linux master control will send (in a future implementation) the LLR encoded data to the VN nodes; in the opposite way, the master receives the output data at the end of the algorithm iteration by the elaboration of the output_mem data.



The other interfaces allow the LDPC decoder Message Passing Algorithm to be executed. They implement the connection with the PHAL, therefore to the other PEs. During the exchanging packets step, through this interfaces, message packets flow from and to the other PEs. The number of virtual data interfaces depends on how much processors the network is composed by: basically there's one interface group for each PE connection, so there will be only one for a 2-PEs network or three for a 4-PEs network.

Each virtual data set of interface pins is composed by 2 data signals of 32 bits (one for input and one for output) and 4 control signals as the figure shows.



The reason why using so many control signals is because signals passing through these interfaces move from and towards fifo registers in the PHAL component, so the control signals are needed to manage those last one. The ASIP always keep controlling the *empty fifo* control signal (*ef*): if goes to '0' it means that there's something in the PHAL fifo to process, so it set the *read enable* fifo signal (*re*) and obteins the data through the *virtual data input*; on the other hand, when ASIP's got something to throw, set the *write enable* fifo signal (*we*) and sends the data through the *virtual data output.*
Those elements here presented are complex components formed by simpler objects, such as counters, memories and controlling components. The protocol of reading and writing operations performed follows a recognized standard of communication: a write and a read enable, plus the control on empty or full fifo. The PHAL-based system needs this interface resources since it controls several operations and some registers temporarly storing the incoming informations are compulsory.

5.2.1.4 The automatically performed instructions

In the main designer layer of the LisaTek framework, other fields are mandatory, such as the definition of *main* and *reset*, *fetch* and *decode* operations.

The *OPERATION reset* only initializes every register. The main program must always begin with a global reset, to start the instruction executing from the first one and to clean the pipeline and its registers.

The *OPERATION main* is always performed because rules the correct flowing of the entire main program code, by incrementing the PC register. Indeed, it shifts the instructions contained in the pipeline through its execution and activates the fetch operation, unless the pipeline it's not been stalled. Practically, thanks to the main, the program instructions are scrolled in the order decided by the status registers.

The *OPERATION fetch*, firstly controls if the condition for a branch has been accomplished, in these case giving to the fetch program counter the value of the *Branch Program Counter,* the addressed place where to jump. On the contrary, it charges the *Instruction Register* with the instructions contained in the place of program memory pointed by the program counter.

The *OPERATION decode*, as the name suggests, decodes the instructions and permits its execution. The execution stage of the pipeline is not implemented as a single instruction. The execute is activated during the decode stage to be performed during the following clock cycle. The kind of execution will depend on the instruction fetched by the Instruction Register.

Some operations are automatically performed during the fetch stage of the pipeline to manage the network traffic travelling from a PE to another. They're essential for the correct execution of the algorithm since the normal algorithm processing flow of execution can't be stopped whenever external data is coming. It's essential to keep executing instructions and calculations in the main code flow while receiving from the external world, i.e. the PHAL, a change of status or new data.

In order to accept automatically a new status request, the *OPERATION adquire_state* is performed. Initially, each one of the 2 bits input status is placed in a register, for comfort of use; then, the only thing this operations execute is changing the exit value of the current status in which the ASIP is supposed to be. But the most important thing is that the value of the registers STATUS_OBJ, in which we saved the current status of the ASIP, will influence the normal flowing of the main code. The processes carry out during each step are all in the same program code and the normal flowing isn't interrupted to execute some other operations, such as the initialization. Different routines contain the instructions necessary to perform every stage and their execution is controlled by branches checking the status object registers.

More complex task is performed by *OPERATION adquire_virt_data*: the operation is activated every fetch and keeps on monitoring all the control input signals *empty fifo* related to each virtual data input, waiting for some data in the PHAL fifos to charge in memory.
Depending on which virtual interface is receiving the data, the operation activates the corresponding *read enable* control signal and place the packet in a temporary register.

The received word might be a Q message or a R message and they must be stored in two different memories. But both of them have the same format and the same field division. There's a problem on how distinguish one from the other.
The numeration of the nodes may be helpful, because VN nodes are identified by numbers with a range between 0 and the maximum VN nodes number (2303 in WiMAX case) while the CNs from that number (plus 1, i.e. 2304) till the total of all the VN and CN nodes (it means 2304+1152=3456). For LDPC purpose, it'll be sufficient to look at the word field containing the destination node (between the 6th and the 17th bit) of the temporary register (the input data just adquired). In case the number is smaller then the maximum number reachable by a VN node, sure the element in the field must be a VN node and consequently the input is an R message to be stored in the *CN_to_VN_mem*. On the other hand, if the number is greater, there's a CN node and the data carryies a Q message to be stored in the *VN_to_CN_mem*.

The storing process is not as easy as previously explained, because each VN receives up to 6 messages (and each CN up to 7) every iteration (of a time slot duration). Hence filling one of the 6 memory cells dedicated to the node could overwrite of a message, if it already contains one message previously received. For that reason, it's necessary to take off every cell in the space of memory dedicated to the destination node (*VN_to_CN_mem* in case of CN and *CN_to_VN_mem* for VN) and to control which is the first one empty, then fill it with the new data.
The control is performed by checking if the destination node field (6th to 17th bit) is full of zeros, ie empty or not. In order to be able to achieve this control, it's mandatory to clean all the *VN_to_CN_mem* and *CN_to_VN_mem* memory cells at the end of every calculation (when the new created messages are stored in the CNi_mem or VNi_mem), having no longer need of the contained data.
An even particular case is based upon the VN0 node. For this element the previously described checking method doesn't work, because its identification number (all zeros) corresponds with the same control cryteria (it's all-zeros or not?). To solve this, when node zero is met, a dedicated counter (*VN0_counter*) causes the data to be stored in the very first rooms of memory.

5.2.2 The instruction set

Declaration of the instructions are mandatory for the main field CoWare framework too. The customized instructions are executed in the decode stage of the pipeline and, for their nature, it's possible to organize two groups. One of these is the *branch instructions* where conditional and unconditional branches are declared and activated. In the behavioural part of the instruction the *Address Register* is assigned with the address indicated in the code and the pipe is stalled at decode level, as to jump to the other program memory cell. Plus, since the decode and fetch stage contain instructions that no longer need to be executed, a flush operation is performed to clean the pipe and to start with the instructions of the jump branch.

All the others instructions of similar nature can be fitted in the same instruction set. They only access data memory and perform even complicated calculations.

The feature and aim of the SET and RESET instructions are obvious, since they only give a '1' or '0' value to the bit in the register literally expressed after the instruction sintax (e.g. SET DONE, puts a '1' in the DONE register). Actually, they're the only two composed by the instruction label and an operand. They are used to control the repetition of the messages routines.

The construction of the other instructions is much more complex and articulated, although each one is executed in only one clock period.

5.2.2.1 The main algorithm instructions

Basically, the LDPC algorithm is achievable using only 4 instructions, expressed into the instructions.lisa field of the processor declaration, as it can be seen in the Appendix B.
These are the following::

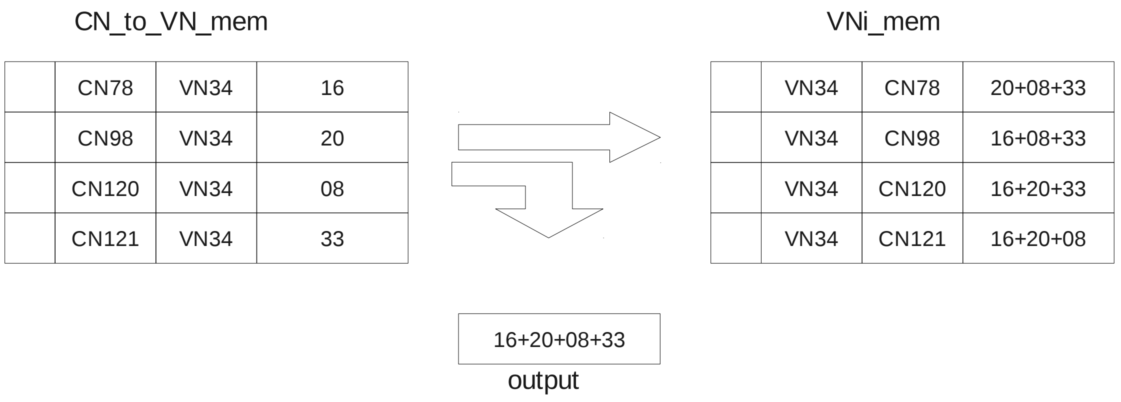
- **Create VN :** creates the Q messages exploiting the data contained in the CN_to_VN_mem spaces of memory and stores its in the VNi_mem memory. Furthermore it performs the output data for each VN node;
- **Send VN :** delivers outside the Q messages stored in the VNi_mem to the CN nodes or, in case the CN target is contained in the same PE of the VN source, copies the packets, orderly by CN, in the VN_to_CN_mem being, careful not to overwrite (it knows it by the VNi_mem word interface field);
- **Create CN :** realizes the R messages creaction using data contained in the VN_to_CN_mem memory and stores its into the CNi_mem;
- **Send CN :** forwards to the VNs of other PEs the R messages stored in the CNi_mem or, in case the node VN target lays in the same PE of the CN source, copies the packets, orderly by VN, in the CN_to_VN_mem, being careful not to overwrite.

The instruction set is been chosen as only one instruction can perform the processes of creating Q and R messages and sending packets. Instructions don't need any extra label because all the information is mapped in memory. The checking constants too are already contained and the counter variables mantein their value because of its global nature. Therefore each one of all the full algorithm computations is done by one of these instructions and a branch operation checking the DONE register and creating a repetition loop. The DONE register will be set internally to each operation at the end of the all the needed iterations.

5.2.2.1.1 Create VN

It's useful to remember how the VN nodes work: what's their task and how they use and manipulate the data to create the output data and the Q messages for the CNs connected to.
Let's suppose having a VN connected to four CNs, so containing four 6-bits datas in its *CN_to_VN_mem* (in practrice four words with the pair CN-VN, data and interface number), representing the R messages previously come from the CNs. The output data will have to be stored in the output_mem, at the location with address equal to the VN identification number, as the sum of **all** the four datas. The Q message for a CN is the sum of all the datas, **except** the one previously arrived as R messages having CN as source node.
These Q messages will be stored in the *VNi_mem*; later the *send_VN* instruction will deliver the packets to the destination interfaces indicated in the interface field.
In tabel, there's an example of messages and output creation.

The instruction is executed a number of times equal to the number of VN nodes in the PE, plus all the needed accesses to the memory, only one per clock cycle. At the end of all the iterations, when all the VNs are scanned, the DONE register is pulled up and an external branch control allows the program flow to continue with the others istructions.



CN_to_VN_mem

	CN78	VN34	16
	CN98	VN34	20
	CN120	VN34	08
	CN121	VN34	33

VNi_mem

	VN34	CN78	20+08+33
	VN34	CN98	16+08+33
	VN34	CN120	16+20+33
	VN34	CN121	16+20+08

16+20+08+33

output

The behavior is very simple. During the first iterations, datas of the same VN are saved in a registers vector, and so the destination CNs at the same index, but in other vectors ; this is done because it's not sure that the connections are equally ordered in both the memories (e.g., for the previous case, in the same raw there could be the connection with CN78 in the CN_to_VN_mem memory and the connection with the CN98 in the VNi_mem one).
When the registers vectors contain all the useful information, the instruction consists in summing all the contents for the output and, by a control of the CN index, summing all the datas except the ones in the same CN for the Q messages.

The results of the sum operations are stored in the *VNi_mem* in the correct order and so the next instruction will be able to send the words orderly.
After each VN node message calculation, all the respective 6 cells in the *CN_to_VN_mem* are cleaned to avoid a bit field confusion during the data adquisition stage.

5.2.2.1.2 Send VN

The sending operations would be easier if the nodes origin and destination weren't allowed to be implemented in the same PE. Since this is not only possible, but even very frequent, we have to consider that the message could be delivered to an inner node, hence directly stored in the VN_to_CN_mem (or the CN_to_VN_mem in case of VN) instead to be thrown out through any of the virtual interfaces.

The task of *send_VN* instruction consists on scanning all the entire VNi_mem, where the Q messages were stored at the end of the calculation steps concerning the *create_VN* instruction.
Firstly, it will control if the memory word contains something (it could contain an all-zeros word in case the VN node does not reach the maximum number of connections) by checking the CN destination field. Then it will get the interface number.
If the binary number contained is a number between 01 and 11, it's directed outside to another PE through an external virtual interface. So the ASIP deliveres outside the word by setting the write enable of the virtual interface indicated by the interface value and places the word in the virtual data out bus. If the interface field shows a double zero value (00), the CN which the message is destinated to, lays in the same PE of the VN source: the ASIP must store it in the *VN_to_CN_mem* ordered by CN number.

As the adquire_data instruction showed, other messages could have been already stored, so it's essential, to avoid overwriting, to extract all the seven consecutive words beginning from that CN address, in order to reach the first empty word. In that cell, the word will be stored.

5.2.2.1.3 Create CN

A CN node produces one R message to every VN is connected to, that it's indicated in the *CNi_mem*, while the data to be used is stored in the *VN_to_CN_mem*. The message to one VN is the minimum 6-bits value, to be chosen among all the previous data messages conteined in the *VN_to_CN_mem* (in the space of that CN), except the one previously received by the VN which the message is addressed to.

For every CN, the program will pull out, scanning all the *CNi_mem,* the identification number of each CN, with the related ordered VN identification numbers and will search in the *VN_to_CN_mem* the data to manipulate. When it will have stored in a temporary registers vector all the data elements, the algorithm instruction will create the R messages, excluding data of the VN to which the message is addressed, by finding the minimum value. Eventually it will store the message packets in the *CNi_mem* and reset the *VN_to_CN_mem*.

5.2.2.1.4 Send CN

The *send_CN* instruction is very similar to the send_VN one, except for the memory; indeed, in this case is the *CNi_mem* to be scanned.
As for the other sending instruction, the control is based upon the interface field. Depending on that value, all the CNi_mem packet words are delivered outside through a virtual interface or stored in the *CN_to_VN_mem*.

Something more here it's the control upon the number of the VN node, because in this case it's possible to meet the zero VN node and the control cryteria could fail. That's why a special counter is dedicated for the VN0.

5.2.2.2 Other ASIP instructions

The LDPC algorithm could be executed even with this four instructions. But for a better performance, other two instruction must be declared. They're not used as frequently as the previous four, but are necessary for the correct realization of the decoding.

5.2.2.2.1 Charge input

Since the *VNi_mem* and *CNi_mem* are filled statically by the main assembler program, there's no need to perform this operation during the initialization procedure of the ASIP.
Neverthless, we can performe other operations in order to simulate this stage; the idea is filling the *input_mem* with the original LLR data originally contained in all the VN nodes implemented in the PE.

The task is achieved by scanning the *VNi_mem* (at the beginning, when no operation has been performed and no data has been modified yet, all the data contained in the six least significative bits is the original input data), jumping six memory cells at a time (for all six cells there will be the same VN, then with the same LLR) The instruction will take the 6-bits data and store it in the input memory, one per cell, in exactly the *VNi_mem* memory order of VN.

Performing the LLR charging instruction and the four LDPC algorithm instructions, the decoding iteration to reach the correct data output can begin and may continue for an indefinite time, as the Linux master control decide. At the end, when the STOP status will be evocated, the program flow will exit from its normal routine jumping to the last step of the algorithm: the delivery of outputs.

5.2.2.2.2 Send output

After that every iteration, such the exchange of messages between nodes VN and CN, have been repeted the amount of times decided by the Linux controller, the output memory is filled with a 6-bit data for each VN node (the two most significative are unused, so set to '1'). These are the error-free data that the LDPC decoder gave us, but still it's necessary to extract the real result data without redundancy. Only 1 bit, which the 6-bit output data is the representation of: the sign bit, i.e. the most significative one.

The *send_output* instruction throws out this bit through the *Virtual Data 0* interface, that's thought to be connected directly to the Linux controller. Since the memory is accessible only one time per clock cycle, the single bit could be sent ones per clock cycle, but this would be a waste of bus space, knowing that the interface is able to carry a 32-bits word.
For that reason the instruction, scanning all the output memory, every period take out one bit from the byte in a cell to put it into a 32-bit word, ordered from the least to the most significative bit. When it has filled all the word, reaching the 32 output data bits, it delivers the packet through the virtual data 0 interface.

When it has thrown the bits of all the VN nodes, it sends the last words (filling it with zeros if the number of VNs is not a multiple of 32 and some word bit remained empty) and sets the DONE register to exit the operation loop and to rest in the final infinite loop zone.

5.2.3 The ASIP main assembler program code

The main program code of the ASIP component, as Application Specific Instruction-Set Processor, is build of the costumized instruction set described in the previous section. In other words they represent the bricks of the program code.

The flow followed by the processor to achieve the LDPC decoding task lays in the appendix C (progr_main.asm).

5.2.3.1 The C code and the loading connections up

The first two groups of words provide the statical charging of the VN and CN nodes connections. As already said, the VN nodes connections carry in their data field the LLR 6-bits input relative to that particular VN, while the CN nodes connections conteins only an all-zeros 6-bits frame in the data field, since they contain nothing at the beginning.

The very first raw means the section *data_VN* in writing mode is declared. This label is evocated in another file, the *progr_main.cmd*, about which it'll be discussed in the next section. Basically, every word written below that section declaration will be stored (in the one row per cell format) into the *VNi_mem* memory.

The hexadecimal word is due for comfort to express into an 8 values number what would require a 32 bit information width. It's clearly composed by all the fields described in the ASIP memory section.

The same format is followed by the *CNi_mem* charged words, now evocated by the *data_CN* label.

When facing an architecture as the WiMAX LDPC algorithm, containing thousands of nodes, each one with a number of connections up to seven, the number of hexadecimal words to build grow significantly. It's clearly unthinkable to manually write all the connections word from the indication of nodes, data and interface.

The WiMAX has well-known LDPC decoder algorithm features, such as the H matrix reporting the connections between nodes and the encoded LLR information. Merging all of them it's possible to obtein several files containing all the connections and data information needed:

- **VN interconnections files :** they show each VN node by its identification number and all the connections to the CN nodes (by identification number too). Plus, they indicate the VN and CN nodes processor membership, so if the message should be delivered outside or just stored in the same PE origin;
- **CN interconnections files :** having the same format of the VN one, describe all the interconnection of each CN;
- **input file :** carries the LLR input 6-bit data information, which each VN is responsible of. Each row conteins a 6-bits word representing the significative bit, 2304 as the number of VN nodes.

The availability of these resources, organized has formatted files, encouraged the creation of a simple C-based programme able to implement a complete list of interconnection hexadecimal words by interpreting the files.

The assignment part of the VN and CN field is pretty easy, just a conversion to a binary value; and so for the interface field assignment, knowing which interface connects one PE to other PE.

Concerning the VN interconnections files, the relative VN input charging is quick, since in the input file, the LLRs are ordered by increasing VN identification number.

Finally, since the LDPC algorithm's been working on is an irregular one, it's not exactly known the number of interconnection words created for a single VN or CN. But knowing its upper limit, it's easy to make the C program instantiate an all-zeros word for the cells lacking.

5.2.3.2 The core of the program

After the memory initialization, the real program algorithm code begins (marked by the label *.text*). Since this processor is a Special Purpose one, the instructions were totally built for the LDPC aim in a simil-C way of programmation, although the code is an assembler-like one.

At the boot of the program, hence of the PHAL, this is supposed to begin from the STOP status, provided in the reset operation and indicated by the 00 pair of bits in the new status.

The 2-bits input responsible of the new status changing process, sent by the PHAL to the ASIP, are immediately stored in two status object registers, one bit per register. In this manner it's easier to control the evolution of the program code, i.e. the execution flow. The code keep on checking, through the branch instructions, the contents of the *STATUS_OBJ0* first (bit 0) and then of the *STATUS_OBJ1* (bit 1). Knowing the correspondence between the status and the 2-bits value representing it,

STOP status	->	***00***
PAUSE status	->	***10***
RUN status	->	***01***
INIT status	->	***11***

is possible, through single bit check, to orientate the flow.

The bit value of the current status was decided depending on its nature. In fact, it's possible to divide the status into two groups: STOP and PAUSE, having the least significative bit '0'; RUN and INIT, having the least significative bit '1'. This separation make easier to control the flowing, since STOP and PAUSE status have similar behavior than the others. They both have to wait for something changing the status (STOP only during the first phase), while the RUN and INIT perform always an instruction. Since only one bit of the status registers is controlled at a time, it's possible to wait or immediately look at the other status registers bits.

The first instruction is a branch control operation (*BC*), jumping to the address indicated by the label *_loop* if the contents of the register is equal to '0' (*EQ*). Indeed, for how the relationship status-bits was designed, if bit 0 is '0' value, it might be only STOP or PAUSE status request and the program should stay there doing nothing but waiting a change of status.
When bit 0 had turned to '1' value, the program follows checking the other bit register, because the status may be RUN or INIT. The next instruction is another branch control one, but this, checking bit 1, must jump if it conteins a '1' value (*NEQ*). In that case, the status would be a “11”, i.e. INIT status. The flow jumps to the initialization routine where it'll have to perform the very first operation.

Into the initialization routine, only one instruction is executed, the *charge input*. Since the LLR inputs and the connection indications are alredy stored in the memories, this operation only take the input data associated to each VN, stored in the 6 least significative bits of the words in the *VNi_mem*, and put it into the *input mem*. In this manner, it will be a directed VN-based ordered correspondence between the *VNi_mem* and the *input_mem* (only the *VN_to_CN_mem* and the *CN_to_VN_mem* are ordered per increasing CN and VN respectively).

The execution of this instruction is internally controlled by a counter, checking the scanning up of all the VNs in the PE. When the counter will reach the total number of VNs, the register DONE will be switched to '1' value, internally to the instruction, and eventually the external branch control will make the program flow exit the routine and return to the main program execution.

Filled the input memory with the original LLR data, all the instruments to begin the LDPC message passing algorithm calculation and sending operations are ready and the main flow can be performed.
When the PHAL sends the RUN status order of execution, the program flow is able to go waiting for the ENABLE signal that fixes the beginning of each time slot.
Before to proceed with the task execution, it's necessary a control of the status, that has possibly been changed during the initialization procedure: the algorithm could have been paused or already stopped for any kind of problem. It's mandatory to control the status to assure is RUN.
If the status was turned to PAUSE, the program will wait there till the RUN command or, in case of a STOP command, will go to finalize the execution to the sending output routine. Normally, the RUN status is guaranteed at least for the first iterations, obviously, so the flow will proceed with the

next step.

Initially, the ASIP should build the new Q messages merging both LLR inputs and R messages received during the previously phase; but since the ASIP haven't received R messages from others CN yet, being at the very first step, the Q message data is composed exclusively by the LLR input, the same by each VN to all its CNs.
For this reason, just the *send_VN* instruction will make part of the first iteration, since the LLR inputs, hence the first Q messages, are already contained in the VNi_mem connection words. The instruction reads the interface field and sends each word through the corresponding virtual interface or, in case it reveales a “00” value, stores it, ordered by CN, in the *VN_to_CN_mem*. The instruction termination is, like before, controlled by the DONE register, set when the number of VN contained in the PE will be reached.

After the sending VN message step, the program will stand in an awaiting state, during which the messages from the others PEs, directed to its CNs will have the possibility to arrive and to be automatically stored in the memory. Till the ENABLE input will be set to '1' other time, in order to advice the beginning of another time slot and than the execution of another task.

Before to begin a new time slot and so to perform a new task, the program must be sure to stay still in the RUN condition of operations.
The next step, each CN contained in the PE has to process all its messages to create the R messages, that will be sent to other VNs. Scanning the data in the *VN_to_CN_mem*, each CN creates a different message for each VN is connected to and stores it in the correct word in the *CNi_mem,* till the last one CN.

After this operation has been totally performed, the algorithm, through the send CN instruction, scans all the *CNi_mem* and sends all the word messages through the virtual interfaces or stores the packets in the *CN_to_VN_mem*, in case that the VN to which it's directed is contained in the same PE.

In the awaiting step of the following time slot, the R messages directed to the VNs will have been arrived. When ENABLE signal will be set, the program flow processes the instruction producing ouputs and Q messages (*create VN*), that will be stored in the *VNi_mem*. After that, through the *send VN* instruction, all the messages are delivered away or stored in the *VN_to_CN_mem*. An uncondinated branch at the end will restart the iteration from the CN node procedure task.

The P_HAL lets the iteration run for several time slots, just to permit the LDPC alghoritm to converge. Finally, it will switch the status operation to STOP, in order to communicate the ASIP to stop processing data and to throw out results. For this purpose, it jumps to the send output routine, where the single instruction *send output* elaborates data contained in the *output_mem* and sends, through the special *virtual interface 0*, the 32-bits output words to the linux master controller.

Scanned all the output memory, the algorithm flow lays in an infinite loop waiting for a global reset and a new decoding operation performance. Actually, the reset intervention is the only way to quit this routine, because there are no branch control operations inside that. It couldn't exist another decoding process after the first one, since that every memory and every 6-bits data in the VNi_mem should be cleaned. Otherwise, when the initialization status began, the input memory would be filled with random incorrect data.

The execution routine of calculation or sending messages by each instruction is controlled by a branch operation. This means that a part of clock cycles needed by the routine are lost in the DONE register checking up, without doing nothing else. Basically a waste of clock cycle. It might be thought to include within the instruction itself the control of jump, to repeat the operation or move to the next one by directly modifing the Program Counter address. This solution, however, would render the code less flexible to possible changes: as it is constructed now, it's furnished of instructions for execute the algorithm LDPC separated by the branch instructions. The advantage is the possibility to modify the algorithm without changing the nature of instructions.

5.3 Tools: LisaTek, Processor Designer, ISE (Xilinx)

For the realization of the project, we made use of several tools to aid the development of processors and electronic systems in general. In particular, we needed an instrument where we could design every single part of the ASIP device, included the instruction set and the program flow, as a real complex dedicated processor. On the other side, in order to create the PHAL component, we weren't looking for a dedicated flexible processor but on a simple routing element, no matter how heavy.
The first tool used to meet the achievement of the ASIP is the *LisaTek CoWare* tool, which exploits the use of the SystemC language to create every part of the processor. For the component PHAL, it was decided to implement it in VHDL using the *Xilinx ISE* environment tool, as it was then possible to synthesize it in order to know the potentiality of the device.

5.3.1 The LisaTek Coware tool

As already said in advance, the building up of a customized processor ASIP implies the declaration of every single part of the component, as:

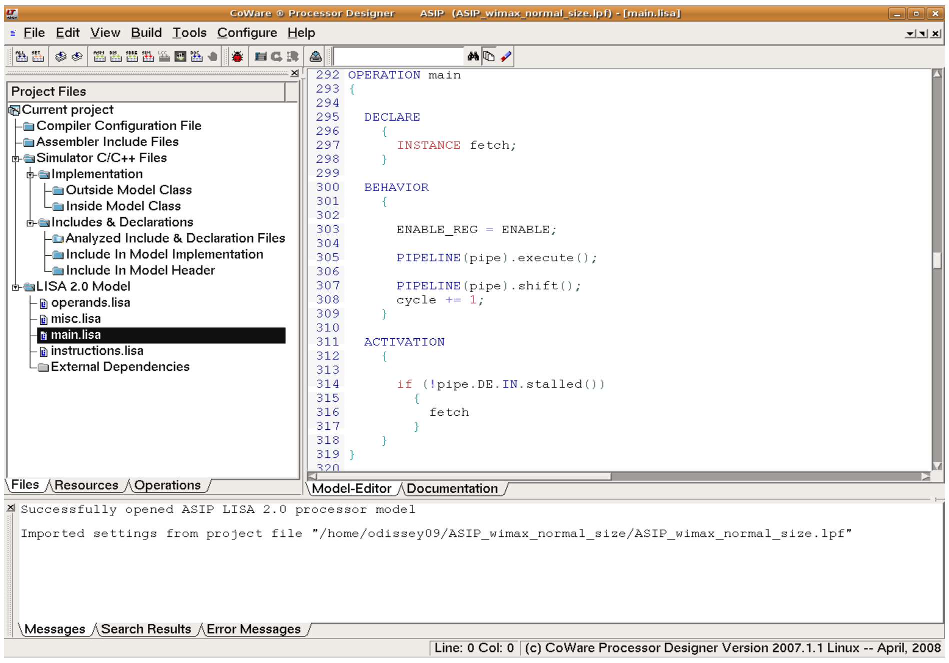
- description of the program code assembler instruction set;
- main program code;
- implemented resources, such as pipeline system, set of registers and memories;
- input and output pin interfaces.

After to proceed with the physical implementation, it's necessary to check the syntax of the created code and then perform the simulation and debugging process in order to control how it works. Eventually would be interesting to synthesize the whole processor, obtaining physical parameters as area occupation, maximum frequency and throughput.

The LisaTek programmer provides for the ASIP purposes several separated frameworks. The first tool a user must work on is the *LisaTek Processor Designer.* It's used for declaring resources and instruction set, checking the syntax and creating the executable files. The following step, the processor simulation, was performed by using the *Processor Debugger* software. Finally, thanks to the *Processor Generator*, the ASIP could be synthesized from the software to the hardware point of view.

5.3.1.1 The Processor Designer

The *Processor Designer* offers an optimal user visual interface. The job is easily splitted in a few ordered sectors and the software helps using the SystemC code language. There's an example of the main view in the figure below.


CoWare ® Processor Designer ASIP (ASIP_wimax_normal_size.lpf) - [main.lisa]
File Edit View Build Tools Configure Help
Project Files
Current project
Compiler Configuration File
Assembler Include Files
Simulator C/C++ Files
Implementation
Outside Model Class
Inside Model Class
Includes & Declarations
Analyzed Include & Declaration Files
Include In Model Implementation
Include In Model Header
LISA 2.0 Model
operands.lisa
misc.lisa
main.lisa
instructions.lisa
External Dependencies
Files Resources Operations
Model-Editor Documentation
Messages Search Results Error Messages
Line: 0 Col: 0 (c) CoWare Processor Designer Version 2007.1.1 Linux -- April, 2008

```
OPERATION main
{

  DECLARE
    {
      INSTANCE fetch;
    }

  BEHAVIOR
    {

      ENABLE_REG = ENABLE;

      PIPELINE(pipe).execute();

      PIPELINE(pipe).shift();
      cycle += 1;
    }

  ACTIVATION
    {

      if (!pipe.DE.IN.stalled())
        {
          fetch
        }
    }
}
```

```
Successfully opened ASIP LISA 2.0 processor model
Imported settings from project file "/home/odissey09/ASIP_wimax_normal_size/ASIP_wimax_normal_size.lpf"
```

The parts, in which the code must be divided, are the main one, the instruction one and the operands and misc parts. Every section must contain some specific fields where to describe instructions and hardware resources, pipelined architecture and automatic executing instructions.

Building all the process, if the sintax turns out to be correct, the LisaTek Processor Designer creates the executable archives following a procedure of assembling and simulation, in order to implement the processor designed.

One problem come out working with the LisaTek device was a library one. The LisaTek version used to implement the LDPC decoder dedicated ASIP, needed libraries of an older version of the Linux operative system installed in the virtual machine, the software was installed on. On the other hand, libraries could not be changed without the other programs used stopped working.

To resolve this problem, both the library packages had to be placed into the operative system. Then, thanks to a libraries switch program, it was possible to switch for a little time (e.g. 60 seconds) the normal libraries to the older ones, every time the compiling of ASIP software was to carry out. At the end of the 60 seconds, the normal version libraries would have replaced the older ones while the building process would have already been over. In this manner, a builded application is obteined to simulate.

Before checking the software behaviour, it's necessary to apply the assembling program. This application, from the assembler main program creates an executable program, done through command line by the instruction:

./lasm progr_main.asm

checking the assembler sintax of the instructions used in the main program code.

Another file must be implemented, as important as the assembler program one: the *progr_main.cmd,* indispensable to declare and allocate the memory spaces, which can be found in the Appendix D.

In the first section, each memory is declared, such as its address of origin, size and byte for cell occupation:

- the origin is indicated in hexadecimal number and represents the address from which that memory is addressable;
- the lenght, in hexadecimal too, expresses the number of single addressable blocks into the

memory;

- the byte field, finally, represent the number of bytes by which each single block is composed (so, as seen, a part from the program memory, the cells of the four memories containing interconnections and data are four byte width, i.e. 32 bit, while the cells of others, conteining only 6-bit data, are a single byte width).

The second section evocates that correspondences between a label and one memory it's been talked about in the previous section. This means that, what's preceded by that label in the assembler program will be stored ordered in the correspondent memory. For example, in the assembler program, the main ASIP code is titled by the label .text, while the *.data_VN* and *.data_CN* are followed by the connection-input words.

The executable assembler program just created must be linked to the *.cmd* file, in order to assure the memory partitioning and data storing, with the command line instruction,

./llnk   progr_main.lof   progr_main.cmd

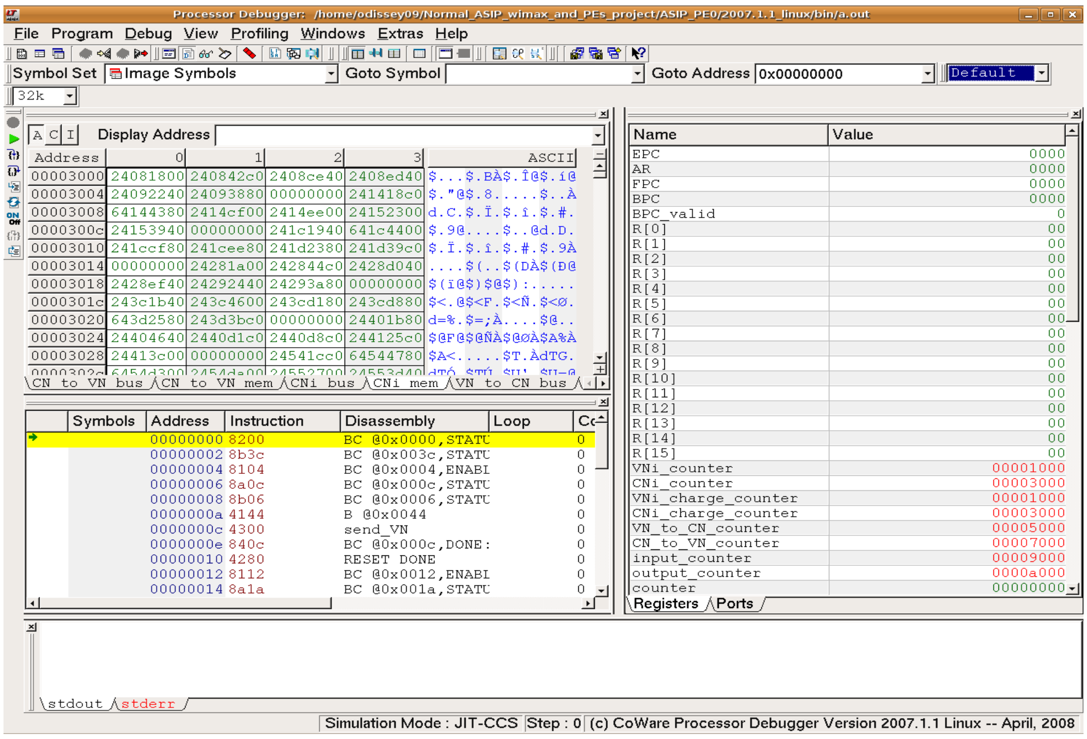
where the *.lof* file is the one previously generated by the assembler code.

Finally, it will be obteined the executable file *progr_main.out,* by which simulate the performed program.

The simulation procedure needs obviously another tool, actually because there's there need to look at the memory rooms and the register contents; the flowing of the program must be controlled and the perfect execution of the jumping operations checked; plus, it's essential to assure the correct functionality of interfaces as well as a good global communication system performance.

5.3.1.2 The Processor Debugger

For this aim, the *Processor Debugger* tool is used.



```
Address          0        1        2        3  ASCII
00003000 24081800 240842c0 2408ce40 2408ed40 $...$.BÀ$.Î@$.í@
00003004 24092240 24093880 00000000 241418c0 $."@$.8.....$..À
00003008 64144380 2414cf00 2414ee00 24152300 d.C.$.Ï.$.î.$.#.
0000300c 24153940 00000000 241c1940 641c4400 $.9@....$..@d.D.
00003010 241ccf80 241cee80 241d2380 241d39c0 $.Ï.$.î.$.#.$.9À
00003014 00000000 24281a00 242844c0 2428d040 ....$(..$(DÀ$(Ð@
00003018 2428ef40 24292440 24293a80 00000000 $(ï@$)$@$):.....
0000301c 243c1b40 243c4600 243cd180 243cd880 $<.@$<F.$<Ñ.$<Ø.
00003020 643d2580 243d3bc0 00000000 24401b80 d=%.$=;À....$@..
00003024 24404640 2440d1c0 2440d8c0 244125c0 $@F@$@ÑÀ$@ØÀ$A%À
00003028 24413c00 00000000 24541cc0 64544780 $A<.....$T.ÀdTG.
```

```
Symbols  Address  Instruction  Disassembly          Loop  Co
         00000000 8200         BC @0x0000,STATU           0
         00000002 8b3c         BC @0x003c,STATU           0
         00000004 8104         BC @0x0004,ENABI           0
         00000006 8a0c         BC @0x000c,STATU           0
         00000008 8b06         BC @0x0006,STATU           0
         0000000a 4144         B @0x0044                  0
         0000000c 4300         send_VN                    0
         0000000e 840c         BC @0x000c,DONE:           0
         00000010 4280         RESET DONE                 0
         00000012 8112         BC @0x0012,ENABI           0
         00000014 8a1a         BC @0x001a,STATU           0
```

Name	Value
EPC	0000
AR	0000
FPC	0000
BPC	0000
BPC_valid	0
R[0]	00
R[1]	00
R[2]	00
R[3]	00
R[4]	00
R[5]	00
R[6]	00
R[7]	00
R[8]	00
R[9]	00
R[10]	00
R[11]	00
R[12]	00
R[13]	00
R[14]	00
R[15]	00
VNi_counter	00001000
CNi_counter	00003000
VNi_charge_counter	00001000
CNi_charge_counter	00003000
VN_to_CN_counter	00005000
CN_to_VN_counter	00007000
input_counter	00009000
output_counter	0000a000
counter	00000000

Simulation Mode : JIT-CCS | Step : 0 | (c) CoWare Processor Debugger Version 2007.1.1 Linux -- April, 2008

This simulator device allows to execute the code step by step, viewing what's happening in the internal memory and registers, plus the pipeline structure and the input/output signals. In this manner can be checked the behaviour of the ASIP before the implementation into the complete scenary of the Processing Element.

5.3.1.3 The Processor Generator

In order to check the real physical properties of the ASIP device, it's useful the use of the Processor Generator tools. Fixing some parameters in the program, as the memory interface configuration and the script generation, a VHDL code is built starting from the original CoWare LisaTek project. The Processor Generator put in a space of memory all the VHDL files representing the ASIP, as written in the other language.

One time that the VHDL code of the ASIP is obtained, it's possible to put all the files in the Xilinx simulator tool and to synthesize the ASIP. When ENABLE signal will be set, the program flow processes the instruction producing ouputs and Q messages (*create VN*), that will be stored in the *VNi_mem*. After that, through the *send VN* instruction, all the messages are delivered away or stored in the *VN_to_CN_mem*. An uncondinated branch at the end will restart the iteration from the CN node procedure task.

5.3.2 The ISE (Xilinx) tool

Other tool's been used in the project is the *ISE Xilinx* architecture generator. Basically, it's a very powerful instrument able to check VHDL code and synthesize it into an FPGA platform.

The project was in part developed by the use of the SystemC programming code, concerning the ASIP component. But the PHAL component was made through VHDL code. Furthermore, the same ASIP programmed and built with the SystemC code, was converted into VHDL to be implemented on a platform. So the Xilinx was used for this purpose too, in order to check and synthesize the main LDPC decoding device.

The Xilinx work environment has many libraries of platform FPGAs. Once you have checked that the code written or generated is completely correct, it's possible to generate the net list and the percentage of area occupation, choosing the desired platform. ISE also gives information on the frequency of operation, in addition to the total number of registers and logic slices.

6 Simulation Scenario and involved Platforms

6.1 Full Flexible Scenary

Field-Programmable Gate Array (FPGA) is the most widely accepted reconfigurable device in the industry. Reconfigurable computing is expected to cope with high performance and low power requirements of actual and future mobile communications devices. Moreover, due to the large diversity existing in Radio Access Technologies (RATs), the aforementioned architectures combine the flexibility of General Purpose Processors (GPP) and DSPs, with the performance and efficiency of dedicated hardware.

The concept of virtual hardware is analogous to the concept of virtual memory in classical computers, i.e. to provide the user an unbounded amount of hardware resource, instead of memory. Traditionally, applications can address a larger amount of memory than physically available in the computer while a run-time execution environment (Operating System) swaps them from and to a slower memory (i.e. disk) in form of pages. The time spent by the system moving data in and out the main resource reduces application performance; it will increase gracefully as more physical resource is available in the system. Maximum performance is achieved when all addressed memory fits in the system. On the other hand, memory virtualization enables developers to design memory-size-independent applications, i.e. targeting architectures rather than characteristics (memory size), which significantly reduces software production cost.

Therefore, in a virtual hardware environment the user should be able to use more hardware than physically available in the system. This would allow independence between user application and actual device size, analogously to software programs, which can be executed in several computers despite the available memory. Moreover, in actual FPGA designs, designers can not take advantage of new transistors in larger devices because can not exploit the increased potential parallelism without redesigning the whole application.

Hardware virtualization mechanisms are classified in three approaches:

1) **temporal partitioning** maps an application of arbitrary size to a device with insufficient capacity by temporal partitioning the application into smaller parts and running them sequentially. If all parts fit in the FPGA at the same time, throughput will be maximized because one sample will be produced at every cycle;
2) **virtualized execution** specifies applications in a custom programming model which defines some atomic computation called operator or task. An application is defined as collection of these operators and their interactions. The members of the device family implement the abstractions defined in the programming model, i.e. tasks, communication interfaces, synchronization, etc. Each device in a family differs in the amount of tasks that can be executed concurrently, thus in overall performance. This approach offers device-independence for a device family;
3) **virtual machine** maps the application to an abstract architecture. The specific architecture performs the conversion from the abstract architecture during the execution. This mechanism is analogous to the Java virtual machine in software. This virtualization mechanism provides full device-independence.

Hardware virtualization is comparable with traditional hardware abstraction in software contexts. In the latter, software programs can access hardware services from different vendors through standard interfaces. Although the mechanisms in software and hardware are different, the objective is the same: to provide device independence and reduce development effort and cost. As a consequence, the device or set of devices appears to the user as a single virtual device, providing an amount of computational capacity, in terms of

- time-multiplexed **fixed** operations executed by programmable processors;
- time-multiplexed **variable** operations configured in reconfigurable devices (FPGA).

Note that whereas computing resource is time in the former (area is fixed) it is a combination of area and time in the latter; this suggests the complexity of the resource management problem.

6.1.1 Virtualization Requirements

Virtualized environments and abstraction mechanisms usually require of a runtime system embedded in the device. It provides the standard communication mechanisms, synchronization between isolated parts in the device or with external devices and frequently used libraries, specifically implemented for a single device, in order to improve efficiency of some applications (e.g. FIFOs, memories, etc.).

This section enumerates the requirements these runtime systems must provide to the designer.

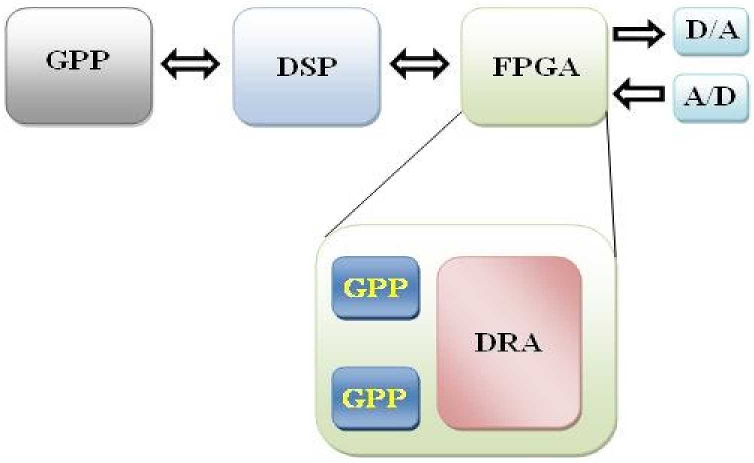
1) **Task partitioning:** Signal processing applications are defined as a set of operations to be applied to a continuous data flow one after another, e.g. codification, symbol mapping, modulation, etc. Thus, this step is straightforward; a natural partitioning might consist on assigning a task to each of these operations.
2) **Resource sharing:** Multiple accesses to the same resource shall be scheduled in time by the runtime environment constrained to application time requirements.
3) **Communications infrastructure:** Each task can have one or more virtual interfaces to send/receive data to/from other tasks. The execution environment must provide efficient routing mechanism in order to deliver data to its destiny under QoS restrictions.
4) **Performance scalability:** Increasing the number of chips or the number of transistors in the chip should translate in a higher performance without recoding or redesigning the application.
5) **Complexity scalability:** Efficient routing, synchronization and management of large designs.
6) **Task mapping:** Tasks are assigned to processors or FPGAs, depending on their nature or implementation language and some optimization goal, i.e. network utilization, load balancing, etc.

6.1.2 Resource Management

One of the most relevant issues in virtualization is the resource management. The proper mapping among the available resources and the required ones becomes a high complex task not only due to the hardware resources management possibilities but also due to the need to find a proper and generic model capable to meet the hardware and software applications particularities. In the side of the available resources we need to talk not only about the available processors, its computing capacity, the memory, the communication links bandwidth, etc but also the scheduling mechanisms and mapping approaches available. Particularly, in FPGA world the available resources are silicon area and time whereas in classical processors it is reduced only to time resources.

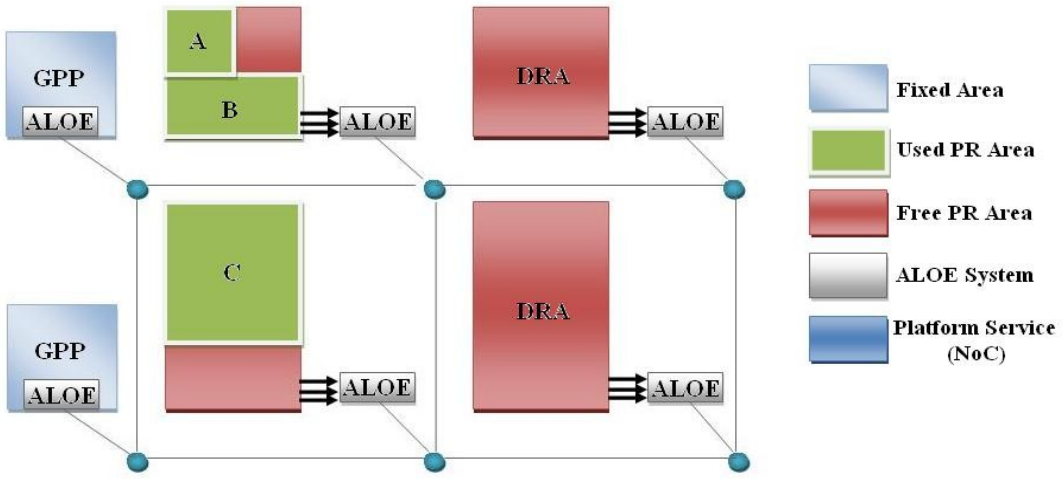
6.1.3 Proposed Virtualized Environment

The proposed platform physically consists on a network of silicon chips. Some of them will be static, in the sense that its circuitry will be fixed, some of them will not. For the former, a program read from a memory might specify the order and set of operations that shall be executed (processor). For the reconfigurable ones, operations must also be defined (bitstream in FPGAs) besides their scheduling, thus, a combination of program and operation-set is needed.



A more general scenario considers a network of mixed devices with static and reconfigurable part, as in the figure above. The designer benefits from the joint computing capacity of all elements. From here and beyond, we will use the term Processing Element (PE) for the static part and Dynamic Reconfigurable Area (DRA) for the reconfigurable one.

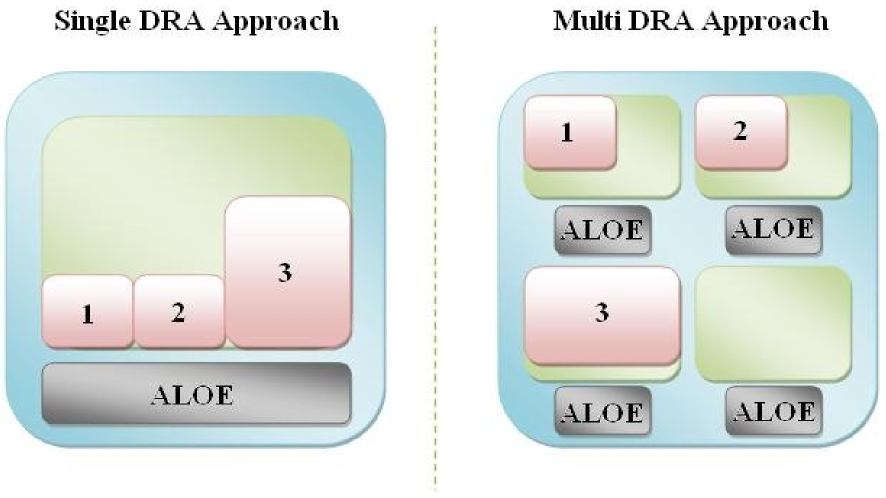
PEs capable to perform context switching (i.e. multi-threaded) will access system-level functions through APIs interacting with background processes (HAL); on the other hand, single-threaded processors and DRAs will use a piece of static logic. The runtime system is called Abstraction Layer and Operating Environment (ALOE) – the term inherits classical PHAL philosophy. This framework, initially designed and validated for software-based computers (PC and TI DSP) , is now being translated to hardware description (VHDL language). Data and control communications, synchronization and execution control mechanisms are abstracted by ALOE which, if present, will use platform-specific services to improve efficiency, i.e. Operating System services, Network on-Chip, etc. Therefore, applications take advantage of particular services without need to know their specific architecture and interface.



A directly acyclic graph (DAG) defines the set of tasks and connections describing the application. Time is divided in slots where the execution of every task is scheduled in a pipelined fashion. Minimum time slot duration has to be equal or greater than the maximum task delay plus the data propagation time (for that task); total application delay will be equal or lower the number of tasks times the slot duration. In order to increase throughput, parallelizable tasks can be executed concurrently reducing slot duration; conversely, those tasks not using the whole time-slot can be multiplexed in time and properly scheduled which reduces area utilization (). Both resources, time and area, can be exchanged matching application requirements. The complexity of this joint management strategy is simplified by relaxing real-time scheduling through the pipelined execution. Therefore, we advocate that conventional signal processing applications for communications are easy to be pipelined due to their data flow nature thus, increasing the potential parallelism.

This advanced resource management is necessary since applications have not been designed ad-hoc for the target platform. Therefore, designers can not optimize time and area; this job has to be done by the runtime system (which has been specifically parameterized for the target device). The pipelined pattern imposed by ALOE allows filling resources in both dimensions.

The problem of dynamically placing and routing components in a DRA is very complex to compute algorithmically. If the DRA is too large, the algorithm can take too much time to be executed in runtime; moreover, a situation of unfeasible or inefficient routing is highly probable after a large number of reconfigurations. Therefore, another approach divides the whole DRA in a set of isolated and closed surfaces where tasks are mapped into, minimizing certain metric (i.e. network utilization). Smaller areas make easier to place and route circuits, moreover, the platform can use specific fixed routing mechanisms (NoC) improving overall efficiency.



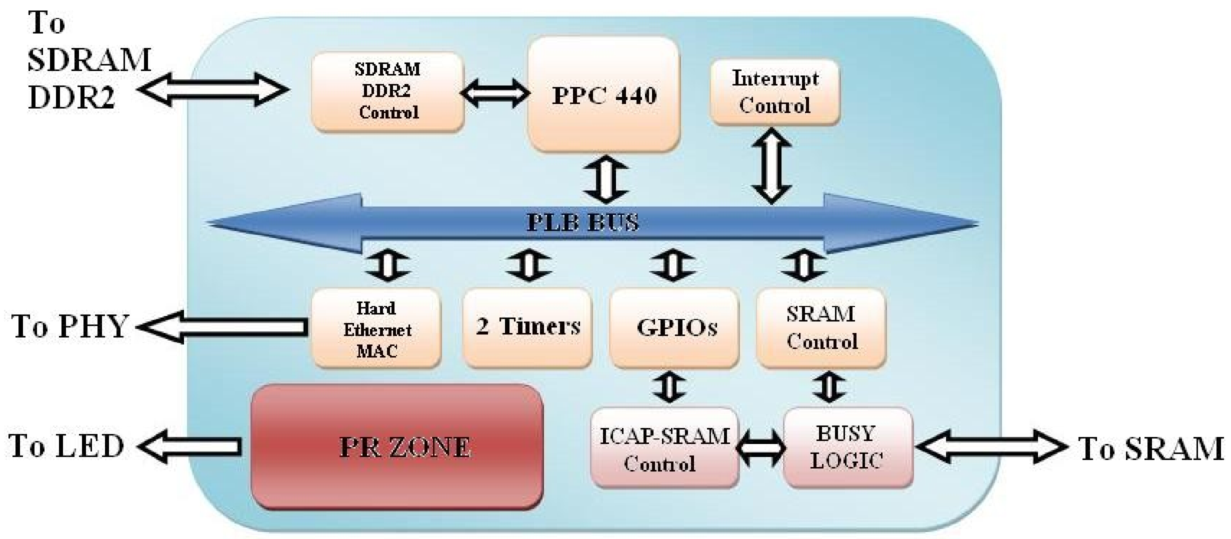
6.1.4 Current Development Status

The platform chosen to implement a preliminary design of the environment has been the ML507 board with a Virtex-5 FX70 FPGA from Xilinx. With the XPS tools one can customize memory and peripherals for the embedded PPC440 running at 400 MHz. The rest of the system has been set to run at 100 MHz. The initial scenario considers a single DRA, configured by a control circuit which reads bitstream data from an external SRAM. Bitstreams are downloaded to the SRAM through an Ethernet 10/100 interface using the TCP/IP protocol stack.

This kind of protocol, although is highly standardized, extremely easy to use and flexible, shows significant drawbacks for reconfigurable computing:

1. it needs about >800 Kbytes of memory to fit a minimal set of functionalities (lwIP stack at RAW mode, no CRC, default buffer sizes);
2. the average throughput it exhibits is significantly below the bandwidth offered by Ethernet. The reason to choose this interface is to integrate, in the next scenario, a hardware ALOE controller with an external software-based ALOE controller running in a Linux machine. Thus, reconfiguration procedure will be fully integrated in ALOE.

Protocol stack memory requirements are too high to fit in the internal BRAM memory of size 64 KB. Consequently, an external SDRAM DDR2 memory was used. In order maintain acceptable performance levels, memory architecture caches 32 KB of program and 32 KB of data from the external memory.



The figure shows the processor architecture: PLB bus is used to access the hard-coded Ethernet MAC controller and the external GPIO ports interfacing the reconfiguration control logic and the SRAM memory (where bitstreams will be saved).

Partial reconfiguration is performed using the ICAP primitive which can read 32 bits of data per cycle. Despite bitstream downloading time through TCP/IP is relatively long, the reconfiguration time is much shorter; consequently, once the new bitstream is loaded in the SRAM, the aforementioned time multiplexing of circuits in pipeline stages is still valid. The table below shows the measured throughput of both situations as well as the reconfiguration time for a 100 Kbytes bitstream.

Transmission	**Throughput**	**Time (100 Kbytes)**
PC - SRAM	*16 Mbps* (mean)	*6,25 ms* (mean)
SRAM – ICAP (reconfig)	*3,2 Gbps*	*31,25 usec*

Area occupation is slightly higher than other approaches, mainly due to the SDRAM and Ethernet controllers. We argue that these resources might be used anyway for another purpose. Only a portion of these devices is used to perform the actual reconfiguration thus, the total overhead is minimal. Table below shows area resource occupation of the design.

	Used	**Utilization**
Registers	*6,232*	*13%*
LUTs	*6,094*	*13%*
IOBs	*211*	*32%*
Memoey (KB)	*900*	*16%*

6.1.5 ALOE-based flexible LDPC decoder

This section describes the implementation of our LDPC decoder using ALOE in the previously described NoC. In order to get benefit from the potential parallelism of the algorithm (computation in nodes) a network of Processing Elements (PE) has to be considered, Network On-Chip (NoC). Efficiency is exploited in both parts: PE and NoC. Nevertheless, the purpose of efficiency usually goes in decrement of flexibility, i.e. ad-hoc designs optimize certain code or a standard family. These designs set PE and network channel capacities to fit loads given certain Tanner graph. On the other hand, a flexible decoder should be able to decode any kind of code, thus any node graph; this means that PE and network need to support any load.

The aim of this project is twofold: to prove virtual hardware paradigm development and to dynamically adapt NoC routing mechanism when code is changed.

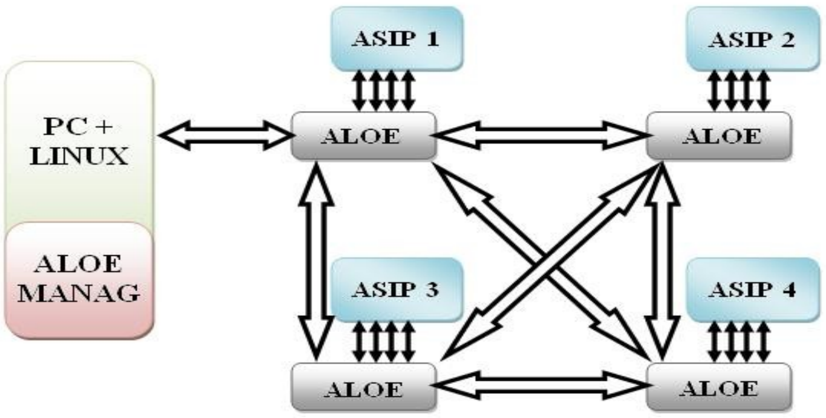
ALOE will be used to abstract platform-specific network interface (NoC), such that there will be no knowledge of it at PE design time. Therefore, NoC routing infrastructure will increase global efficiency albeit the only interface PE designer has to consider is ALOE.

The decoder consists on 4 Application-Specific Instruction-set Processors (ASIP), 4 hardware-based ALOE controllers and 1 PC-Linux running software-based ALOE managers. Considering ALOE time-slotted execution pattern, each ASIP will be assigned a variable set of nodes (*CNs* and *VNs*) which will perform their computations on each slot and so to lighten the work. PEs have an input virtual interface and three output virtual interfaces; one towards each other ASIP. Interfaces are FIFO-like while their implementation type – BRAM, register-bank, etc – resides under ALOE control. Packets through virtual interfaces are merged into the physical NoC channels thus exploiting its efficient communications mechanism; routing policy between PE is maintained, again, by ALOE.

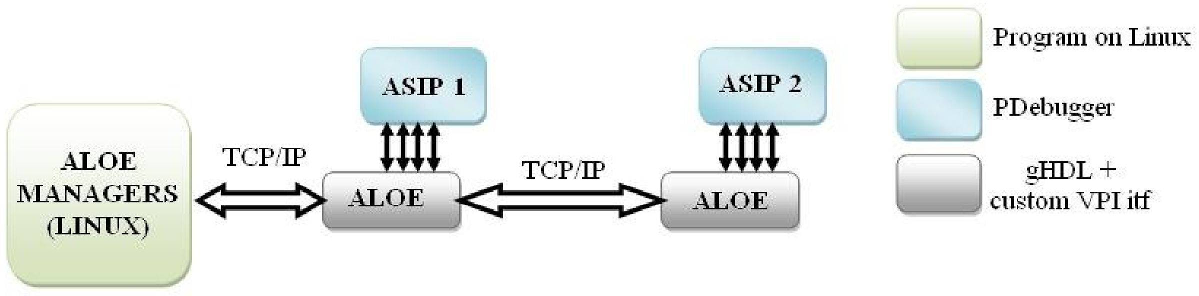
The algorithm ALOE uses to map functions into computing resources is the *tw-mapping algorithm*. This algorithm is inadequate for *Tanner Graphs* because they are not Direct Acyclic Graphs (DAG). Therefore, code selection is performed off-line while an external tool provided in computes node to ASIP mapping and routing.

Finally, simulations have been performed in order to prove the correct behavior. The scenario merges software and hardware (VHDL) descriptions, thus, VPI interface and custom code has been chosen to interface gHDL (GNU tool), LisaTek Processor Debugger and ALOE in Linux.

Due to a license problem, we could not charge the LisaTek program in more than two machines, so we cannot start the whole system simulation.
To avoid the license problem and to allow the verification of the LDPC algorithm behaviour, we implemented a more simple 2 PEs network scenary as in the figure. There were no big problem in changing the main component features, because thanks to how the architecture and the codes were designed, there were actually no need of changes.



The PHAL doesn't really know if or if is not connected to the PHAL of another PE: it only delivers the packet through the external data interfaces without knowing what is on the other side. In this case it's the external Linux master controller having the duty to check that the PHAL send data only by the interface connected to the correct PE. Likely, the ASIP instructions and memory space are the same whichever were the number of PEs connected in network (up to 4, since the interface field in the data word is 2-bit width), because there's no previous indication fixed in the architecture of which nodes are thought to lay in that ASIP.



The only things that had to be changed are the precharged connections-input words, because now the VNs and CNs are splitted onto only 2 processors; so the amount of charged nodes for each PE is now theoretically doubled and plus only one interface (a part the 00) exists, through the other PE. We had already developped a C program to create the interconnections-input word to be stored in the ASIPs memory, able to interprete the identificative number of VN and CN and to assign an interface number according to the direction. So we immediately managed to change a little the program and so to obtein the new words.

6.2 Following the steps for simulation

For comfort of use, instead of taking the loading of PEs on two separate machines, it was decided to charge the PE applications (ASIP + ALOE) on two virtual machine frameworks in the same calculator.
Firstly, it's been necessary to check if the machines were able to communicate by the intranet. During the development of this configuration came out had some problems too, due to the LisaTek license: the machine couldn't get connected and the connection MACs had to be the same, otherwise the LisaTek Processor Debugger did not work.

Once they reached the connection, it was possible to getting started the simulation. The simulation procedure consisted only in giving the change status order by command line, that the linux master controll concerned to deliver to the PHAL and this to the ASIP. For do this a correct order must be followed:

1. **phload tst :** it's the first command, by which the application is loaded into the ASIP, i.e. the program it must follow, with the connections and input words;

2. **phinit tst :** after the application loading, the first operation is the initialization stage, during which the input data is stored in its memory. This might take a long time to execute, since the PEs involved are only two and each one conteins a great amount of VN nodes, so inputs;

3. **phrun tst :** as already said in the main ASIP program chapter, the run command can be launched even during the initialization step. When finally this one will be over, the ASIP will begin the algorithm execution;

4. **phstop tst :** this is the last command to be executed, when the program was already left running for enough time. Finally, the ASIP will throw out, through the virtual interface 0 connected to the master controller, the output data resulted from the message passing iterations.

An important parameter that must be considered is the *time of simulation*; in fact, the flow of the main program can be controlled, but anyway its execution will stop after a time decided by that factor, whatever the program is doing. For this reason is important to set the time of simulation quite large, enough to permit a satisfactory simulation with a reasonable number of iterations and the sending output stage.

Opening the gtkwave wave viewer is then possible to see the signals going to or coming from the ASIP and the PHAL. Furthermore, there's the option to open the LisaTek Program Debugger too and to see what's happening into the ASIP (in the memory and the interfaces) during the program execution, step by step.

6.3 The Xilinx Virtex-5 Family

The Xilinx Virtex-5 family offers the newest and more powerful features in the FPGA world. It's composed by five distinct sub-family platforms, each one performing different capabilities and resources for all the possible implementations.
Xilinx created the ASMBL (Advanced Silicon Modular Block) architecture, initially applied to Virtex-4, to enable rapid and cost-effective assembly of FPGA platforms with varying feature mixes optimized for different application domains. This new architecture breaks through traditional design barriers by eliminating geometric layout constraints and enhancing on-chip power and ground distribution by allowing power and ground to be placed anywhere on the chip.

The Virtex-5 family introduced the first FPGAs produced through a 65-nm triple-oxide technology process. Acting as an excellent programmable alternative to the ASIC customed way, they require the best solution for high-performance logic, DSP or embedded systems designers. Thanks to the revolutionary manifacturing size, obtaines over the previous generation increased by 65% in logic capacity and 30% in speed.

The board could be seen as the union of Configurable Logic Blocks (CLB), made of two slices conteining each one four function generators, configurable as real 6-input LUTs or dual 5-input LUTs, four storage elements, several arithmetic logic gates and large multiplexers. Alternatively, the CLB shall be organized as 32-bit shift registers or 64-bit distributed RAMs. In addition, as for the SelectIO IOBs, the storage elements can be configured as flip-flop D or latches.
The amount of resources is considerable: 32-bit shift registers and up to 330,000 logic cells (depending on the sub-family type), including flip-flops with clock enable and 6-input LUTs.

The system clocking is generated from up to six Clock Management Tiles (CMTs), each one containing two Digital Clock Manager (DCM) and one PLL blocks, for the frequency synthesis and clock phase shifting. The system allows to optimize the low-jitter clocking and to avoid duty cycle drifts by a differential tree-structure clock and 32 global clock networks, by which reaching the frequency of 550 MHz.

RAM blocks provided as true dual-port at 36 Kbits could be programmed from 32K x 1 bits to 512 x 72 bits, in various depth and configurations. This means a total amount of integrated block memory of 16.4 Mbits. Optionally built-in error correction circuitry and enhanced programmable FIFO logic availables.

To interfacing with the external world, parallel SelectIO technology is provided with a width selection of I/O standard operations from 1.2V to 3.3V voltage level, up to 1,200 I/Os ports. Exploiting the ChipSync technology, the interface will be able to communicate in a source-synchronous mode. In advance, termination are furnished by Digitally-controlled impedances (DCI). Their nature makes possible to compensate temperature and voltage variations and easier the board layout, by reducing resistors and placing terminations in the ideal location.
The SelectIO Input/Output Blocks (IOBs) are programmable and its registers could be edge-triggered D-type flip-flops or level-sensitive latches.

High-performance calculations are performed through the use of advanced DSP slices (up to 640), based on 25x18 bit two's complement multipliers and optional adder/substractor, to perform complex-multiply or multiply-adder operations. A 48-bit accumulator for multiply-accumulate (MACC) operations can be provided. Available pipelining and bitwise logical functionality.

In order to assure intellectual property security, Virtex-5 implements in its FPGAs the 256-bit AES bistream decryption, with improved error detection/correction capability.

6.3.1 The XC5VFX70T component

The FXT Virtex-5 platform's been used, assures high performance embedded systems with advanced serial connectivity. It's the only platform of the Virtex-5 family providing PowerPC 440 microprocessors, based on RISC architecture (32 K-byte instructions and data cache included). Working at the very low voltage of 1.0V, it can reach the 550 MHz operation frecuency with more then 1000 MIPS (Million Instructions Per Second) through a multiple instruction per clock cycle architecture, parallelized by a 7-stage pipeline.

Logic Resources	Slices	11200
	Logic Cells	71680
	CLB Flip-Flops	44800
Memory Resources	Maximum Distributed RAM (Kbits)	820
	Block RAM/FIFO (36 Kbits each)	148
	Total block RAM (Kbits)	5328
Clock Resources	Digital Clock Managers (DCM)	12
	PLLs	6
I/Os Resources	Max Single-Ended Pins	640
	Max Differental I/Os Pairs	320
Embedded Resources	DSP slices	128
	PowerPC 440 processor blocks	1
	RocketIO GTX Transceivers	16

Implemented by the TXT sub-family too, FXT conteins RocketIO transceivers: full-duplex serial transceivers capable of 150 Mb/s to 6.5 Gb/s baud rates. A part form CRC generating and checking, it provides programmable equalization systems for the transmitter and the receiver plus receiver signal detection and loss of sognal indicator.

Each platform has a different amount of architectural resources, so the chosen one has the features illustrated in the table below.

What's been generally said for the whole Virtex-5 family applies for the FX70 brench too. So the logic resources will be applied to implement not only the registers but the multiplexers, the comparators and the counters. The ROM containing the program code of the LDPC algorithm will find room through the memory resources, as well as the others RAM memory designed into the ASIP architecture.

All the I/Os resources will have to satisfy the need of I/O pins connecting the ASIP with the PHAL and the Linux controller. Finally, the DSP slices will implement all the calculation part instruments, as the multipliers and the adder/substractors.

7 Results and Conclusions

7.1 The ASIP Resources occupation

The amount of resources requested by the ASIP can be seen in the table. It wasn't possible to get the entire NoC resources occupation, but only of one (ASIP + PHAL) of the two PEs making part of the LDPC decoding network.

Slice Logic Utilization		
Number of Slice Registers	*473 / 11200*	*4,2%*
Number of Slice LUTs	*840 / 11200*	*7,5%*
Number Used as Logic	*840 / 11200*	*7,5%*
Slice Logic Distribution		
Number of LUTs Flip Flop pairs used	*857*	
Number with an unused Flip Flop	*384 / 857*	*44%*
Number with an unused LUT	*17 / 857*	*1%*
Number of fully used LUT-FF pairs	*456 / 857*	*53%*
Specific Feature Utilization		
Number of DSP48Es	*5 / 128*	*3,9%*
Total Amount of Memory (Kbits)	*1120 / 5380*	*21%*
Maximum Frequency (MHz)	*280*	

The use of logic units, including LUTs and registers, appears to be quite small compared to the total amount available in the platform. The exploitation of DSPs for arithmetic-logic calculation is negligible too, since it's about the 3,9%.

As shown in the table below, it's possible to appreciate the memory coverage and the FSM requirement.

FSMs	2	*3-states*	
ROM memories	1	*4K x 8 bits*	
RAM memories	6	*4K x 8 bits*	*2*
		8K x 32 bits	*4*
IO pins	309	*1-bit*	*17*
		2-bit	*2*
		32-bit	*9*

The VHDL language code generated by the Processor Generator from the SystemC original code ASIP description synthesized two Finite State Machine of 3 states each one.

Totally, as expected from the project, the Xilinx synthesizer device declares 7 memories:

- the ROM containing the program code and identified in the ASIP architecture by the program_mem, as a 4K x 8 bits memory;
- 2 RAMS having a 4K x 8 bits size like the program code one, representing the input and output memories (input_mem and output_mem);
- 4 RAMS with the biggest width, 8K x 32 bits, because are the one hosting the interconnection-data buffers, ie VNi_mem, CNi_mem, VN_to_CN_mem and CN_to_VN_mem memories.

Even including the ROM for the area memory occupation, as the total FPGA available is 5320 Kbits, the total needed memory block is 1120 Kbits, about the 21%.

The greatest occupation factor concerns the Input Output pins. In the simulation scenario composed by only 2 PEs, it can't be appreciated because there's only one virtual data interface. But in the general case, with a 4-PE NoC scenario and 3 virtual data interface (plus the virtual data 0) each ASIP, this will keep more signal pins. Eventually, it'll have the consumption of a 48,28% of available pins.

Checking the critical path of the ASIP circuitry, the maximum working frequency reaches the value of 280 MHz. The Xilinx family platforms express the occupation area of the board in number of slices. Through this unit, it's impossible to make a comparison with other solutions, because the number of slices in which a component is implemented depends on the technology.
In order to compare the project with others already present implementation, it's mandatory to convert the slices in equivalent gates. Therefore, it's essential to know the convertion factors for the FX70 in the Virtex-5 family: 1 LUT corresponds to 10 gates, 1 Flip Flop to 4 gates and 1 bit RAM to 4 gates.
For the Virtex-5 FX70 family case, each slice contains 4 LUTs and 4 Flip Flops. Thus, the previous table showed that the slice occupation is 473 slice Flip Flop registers and 840 slice LUTs. Multipling both the slice counts for the number of LUTs and Flip Flops in each slice, it comes to a total amount of 1892 Flip Flops and 3360 LUTs. Through the gate convertion, the logic gate value in table can finally be obtained.

	[14]	**[15]**	**[16]**	**[17]**	**ZO-NOC**	**ASIP NoC**
CN method	phy	3-min	minsum	minsum	minsum	minsum
Precision	8 bit	6 bit	8 bit	6 bit	6 bit	6 bit
Technology	160 nm	65 nm	130 nm	90 nm	130 nm	65 nm
Frequency	500 MHz	400 MHz	83 MHz	109 MHz	300 MHz	280 MHz
Logic kgates	N/A	520	420	380	125	52,52

It can be noticed how the number of gates allocated for the ASIP solution presented is much smaller than any other one in the state-of-art. Considering, however, a network of two processors, such as the implemented one, provides a logic amount of about 105 kgates.
The operating frequency remains quite low compared to other solutions, but it's acceptable

considering the size of the used technology. The most disproportionated resource is the memory, for which there has been no optimization.
It wasn't possible to obtein the real occupation area (in mm^2) and the throughput, due to the lack of the CMOS libraries in the Xilinx synthesizer tool.

How it's showed, the LDPC decoder in the first columns was realized exploiting the original no-linear phy function. This method requires a more complexity, as demonstrated for the major size technology and the 8-bit data information, but at the same time reaches a better efficiency and higher frequency. Neverthless, it has the disadvantage to be able to work only with some custom codes. The other LDPC decoders, though implementing an approximation CN method, have a lower technology size. Among them, the better one is certainly the second, due to the use of a 65 nm technology and the higher work frequency; although the amount of logic Kgates resources is large enough.
All these LDPC decoding architectures have one great problem, having been optimized only for a specific set of codes. On the other hand, the ZONOC (Zero-Overhead NoC) solution, though through a larger technology dimension, is not limited by the code used. Furthermore, it works at a quite good frequency, implementing a low amount of gate resources.

The ASIP-approach LDPC decoder here presented, has features very similar to the ZONOC one. Firstly, it uses the same approximation to the CN node task with the same data precision. The resources occupation is, totally, quite the same and the frequency a bit lower. Besides, its manifacturing technology is the newest 65 nm one. This LDPC decoder, as the ZONOC one, is able to work with different codes and it's not bound by the nature of inputs.

7.2 Limitations and future improvements

The LDPC decoder's been developed through the ASIP approach, could work even with a kind of data input different from that produced by the WiMAX standard. That's because, actually, the nodes aren't physically built in the ASIP architecture as it could be in an hardware way of programmation. The processor doesn't know the existence of the nodes and this for how it's been thought the algorithm. Basically the ASIP read some parts of the packet coming in and compute its to realize some tasks; even there could be more or less nodes and it would be the same.

7.2.1 Parameters limitations

Obviously some parameters have to be respected; for example, the number of CN connected to a VN can not be higher then 6 and the number of VN connected to a CN higher then 7. This is clearly due to the lack of memory space. As a matter of fact, every VN node (and the same for a CN node) has a reserved space in the VNi_mem and in the CN_to_VN_mem memories of 6 word cells (7 in the CNi_mem and in the VN_to_CN_mem for a CN node) to contain all the messages sent by the CNs connected. So if there were more then 6 CN connected to a VN, there would not be enough room for the 7-th message and so on; worst, for how the storing instruction is done, the messages coming after the 6-th to be stored in the same range of memory would overwrite the last cell message, the 6-th.

There are another parameters by which the storing instructions depend, that are the number of VNs and the one of CNs. Indeed, as explained in the ASIP architecture chapter, since the ASIP never knows in which stage the LDPC algorithm is (if the R messages or Q messages computational step), to recognize which type of message arrived (and in which memory to store it), compares a field of the word with the number of VN, because the CN identification numbers follow the

numeration from the last VN identification number.
For that reason it's mandatory to correct these constants into the instruction creation program by manipulating the code just in case the number of nodes are changed.

7.2.2 View to future improvements

The way the LDPC message passing algorithm as been designed within the concept of the Tanner Graph is not the only one achievable. Likely there may be many other possible ways to reach the decoding target.
It might be thought to create in the hardware way the Tanner Graph nodes, thus making even the interconnections between them fixed. But this would make the algorithm static and inflexible to changes in input data and those of the H parity matrix, the value of which depends on the interconnections between the sets of nodes.
Another way to address the problem might be to implement the VN nodes sending a single broadcasting message to all the CN nodes connected to it, containing the sum of all the messages previously received plus the LLR input (the VN output value). The CN node will just have to worry about keeping the message to send to that specific VN node and substract it at the time of receipt in order to obtein its useful information.

Internally to the developed solution, there could be open fields to improve.
It's just been pointed out how , by some fixed parameters depend the storage of data in memory. This is because the total number of nodes affects some comparison operations. But these parameters are contained in registers initialized in the main part of the program and thus made constant.
If it could be possible to write these logs from outside, they could be made variables. The idea is, at the initialization time, to send a word containing the number of nodes for both sets: changing the ASIP code, will be possible to change the value of the two registers and thus make the algorithm adaptive to the number of nodes (ie the number of 1's of the H matrix).

Regarding the employment of memory, what can worry about most is the need of many logic to handle six separate memories. Reducing the number of memories requires a different organization of the individual cells or the implementation of a dedicated extra searching messages algorithm, not too onerous in terms of computational difficulty.
A node could, then, be identified simply by its memory address without having to specify the number into a dedicated buffer field, thereby saving space for more messages in a single cell.

Another point in question just inside the project is the timing and synchronization part. Since the SYNC Deamon block is implemented in the PHAL, it cares to split operations in time slots. But time slots duration is decided in advance, when planning the PHAL and can't be changed during the algorithm execution.
What would happen if the complexity of the algorithm was bigger and the time to perform calculations was too large for the time slot? The code provides that the program flow can't move to the sending packets operation until the calculation of all messages isn't finished. But this doesn't affect the next time slot, because nothing happens if both (or all four) processors terminate after the PHAL has triggered the enable and then started the next time slot. They simply stand waiting until the next step of the enable makes begin a new time operation slot.
The question then arises whether a PE finishes on time and the other not, depending on the distribution of nodes. In this case, one would start a new operation phase while the other would be waiting for the next one, resulting in an uncoordinated receiving packets.
What the Linux master controller can do, then, during the initialization step, would be sending the length of time slot to the PHAL, depending on the complexity of the algorithm. That means an improvement capability for the PHAL, making flexible the time duration of process execution, not just from the controller, but also from the PHAL point of view, who then will check whether the

messages are all sent within the time limit or not and in that case extend the time slots.

The improvement of PHAL functionality is required, since it's easy to join other blocks to the component too. Therefore, a statistical block could be a good way to control data.
However, after adding other parts increasing the complexity, the already implemented blocks should be better fixed. For example, the FRONT END block is the responsible of control command but what it does right now goes beyond its duties. Indeed, it should only check the received packet destination Deamon and forward it to the correct internal block. While in the actual version, it elaborates the data into the packet and sends to the other blocks only few costumed informations.

Concerning the EXEC block, it works exactly as expected. The improvement must be done on the ASIP component. Actually, the ASIP, before receiving a change status request by the EXEC, fills with the corresponding bit value the status registers. It doesn't even control if something goes wrong in the application changing status. What the ASIP could do is a better checking status and responding current status to the EXEC.

In this LDPC implementation, as seen in the main program, the ASIP in the NoC can perform only one decodification. Then, only the reset operation makes quit the execution from the routine loop to begin a new decodification. But, likewise, it shouldn't be possible to decod other data, even with the same Tanner Graph, because the data memories are dirty with the random data produced by the algorithm execution. It should be necessary to clean all the memories up before charging another times connections and inputs.

8 Appendix A (main.lisa)

```
RESOURCE
{
  MEMORY_MAP
  {
    BUS(progr_bus),     RANGE(0x0000, 0x0fff) -> progr_mem[(31..0)];
    BUS(VNi_bus),             RANGE(0x1000, 0x2fff) -> VNi_mem[(31..0)];
      BUS(CNi_bus),           RANGE(0x3000, 0x4fff) -> CNi_mem[(31..0)];
      BUS(VN_to_CN_bus),      RANGE(0x5000, 0x6fff) -> VN_to_CN_mem[(31..0)];
      BUS(CN_to_VN_bus),      RANGE(0x7000, 0x8fff) -> CN_to_VN_mem[(31..0)];
      BUS(input_bus),   RANGE(0x9000, 0x9fff) -> input_mem[(31..0)];
      BUS(output_bus),  RANGE(0xa000, 0xafff) -> output_mem[(31..0)];
  }

  BUS char progr_bus
    {
      ADDRTYPE(unsigned long);
      BLOCKSIZE(8,8);
    };

  BUS bit[32] VNi_bus
    {
      ADDRTYPE(unsigned long);
      BLOCKSIZE(32);
    };

  BUS bit[32] CNi_bus
    {
      ADDRTYPE(unsigned long);
      BLOCKSIZE(32);
    };

  BUS bit[32] VN_to_CN_bus
    {
      ADDRTYPE(unsigned long);
      BLOCKSIZE(32);
    };

  BUS bit[32] CN_to_VN_bus
    {
      ADDRTYPE(unsigned long);
      BLOCKSIZE(32);
    };

  BUS bit[8] input_bus
    {
      ADDRTYPE(unsigned long);
      BLOCKSIZE(8);
    };

  BUS bit[8] output_bus
    {
      ADDRTYPE(unsigned long);
      BLOCKSIZE(8);
    };

  RAM char progr_mem
    {
      SIZE(0x1000);
      BLOCKSIZE(8,8);
      FLAGS(R|X);
    };

  RAM bit[32] VNi_mem
    {
      SIZE(0x2000);
      BLOCKSIZE(32);
      FLAGS(R|W);
    };

  RAM bit[32] CNi_mem
    {
      SIZE(0x2000);
     BLOCKSIZE(32);
      FLAGS(R|W);
    };

  RAM bit[32] VN_to_CN_mem
    {
      SIZE(0x2000);
      BLOCKSIZE(32);
      FLAGS(R|W);
    };

  RAM bit[32] CN_to_VN_mem
    {
      SIZE(0x2000);
      BLOCKSIZE(32);
      FLAGS(R|W);
    };


  RAM bit[8] input_mem
    {
      SIZE(0x1000);
      BLOCKSIZE(8);
      FLAGS(R|W);
    };
```

```
 RAM bit[8] output_mem
   {
     SIZE(0x1000);
     BLOCKSIZE(8);
     FLAGS(R|W);
   };

 PIPELINE pipe = { FE ; DE ; EX };
 REGISTER    short  AR;
 PIPELINE_REGISTER IN pipe {
     unsigned short instr_reg;
     PROGRAM_COUNTER short PC;
 };

 PROGRAM_COUNTER short  EPC;
 REGISTER   short  FPC;
 REGISTER   short  BPC;
 REGISTER   unsigned bit[1] BPC_valid;
 REGISTER uint8 R[0..15];

 REGISTER uint32 VNi_counter;
 REGISTER uint32 CNi_counter;
 REGISTER uint32 VNi_charge_counter;
 REGISTER uint32 CNi_charge_counter;
 REGISTER uint32 VN_to_CN_counter;
 REGISTER uint32 CN_to_VN_counter;
 REGISTER uint32 input_counter;
 REGISTER uint32 output_counter;
 REGISTER uint32 counter;

 REGISTER bit[32] OBJ_REG;
REGISTER bool STATUS_OBJ0;
 REGISTER bool STATUS_OBJ1;

 REGISTER bool ENABLE_REG;
 REGISTER bool DONE;

 REGISTER bit[32] tmp_data;
 REGISTER bit[12] CN_reg;
 REGISTER bit[12] VN_reg;
 REGISTER bit[2] itf_reg;

 REGISTER uint32 VN_temp_count;
 REGISTER uint32 CN_temp_count;

 REGISTER uint8 n;
 REGISTER uint8 s;
 REGISTER uint8 m;
 REGISTER uint8 c;
 REGISTER uint32 VN0_counter;

 REGISTER uint32 total_VN_n;
```

```
  REGISTER uint32 total_CN_n;
  REGISTER uint32 max_VN_to_CN;
  REGISTER uint32 max_CN_to_VN;
  REGISTER uint32 min_CN_value;

  REGISTER bit[6] temp_calc[0..6];
  REGISTER bit[12] reg_calc[0..6];
  REGISTER bit[12] reg_home_calc[0..6];

  PIN IN bit[32] virt_data0_in;
  PIN OUT bool virt_data0_in_re;
  PIN IN bool virt_data0_in_ef;

  PIN OUT bit[32] virt_data0_out;
  PIN OUT bool virt_data0_out_we;
  PIN IN bool virt_data0_out_ff;

  PIN IN bit[32] virt_data1_in;
  PIN OUT bool virt_data1_in_re;
  PIN IN bool virt_data1_in_ef;

  PIN OUT bit[32] virt_data1_out;
  PIN OUT bool virt_data1_out_we;
  PIN IN bool virt_data1_out_ff;

  PIN IN bit[32] virt_data2_in;
  PIN OUT bool virt_data2_in_re;
  PIN IN bool virt_data2_in_ef;

  PIN OUT bit[32] virt_data2_out;
  PIN OUT bool virt_data2_out_we;
  PIN IN bool virt_data2_out_ff;

  PIN IN bit[32] virt_data3_in;
  PIN OUT bool virt_data3_in_re;
  PIN IN bool virt_data3_in_ef;

  PIN OUT bit[32] virt_data3_out;
  PIN OUT bool virt_data3_out_we;
  PIN IN bool virt_data3_out_ff;

  PIN IN bool ENABLE;

  PIN IN bit[2] new_status;
  PIN OUT bit[2] current_status;

  PIN IN bit[32] time_stamp;

  long cycle, instruction_counter;
}

OPERATION reset
```

```
{
  BEHAVIOR
    {
      int i;
        for (i = 0; i < 16 ; i++)
            {
                  R[i] = 0;
            }
      AR = 0;
      cycle = instruction_counter = 0;
      BPC = 0;
      BPC_valid = 0;
      EPC = FPC = LISA_PROGRAM_COUNTER;

      STATUS_OBJ0 = 0;
      STATUS_OBJ1 = 0;
      ENABLE_REG = 0;
     DONE = 0;

      VNi_counter = 0x1000;
      CNi_counter = 0x3000;
      VNi_charge_counter = 0x1000;
      CNi_charge_counter = 0x3000;
      VN_to_CN_counter = 0x5000;
      CN_to_VN_counter = 0x7000;
      input_counter = 0x9000;
      output_counter = 0xa000;

      VN_temp_count = 0;
      CN_temp_count = 0;

      s = 0;
      m = 0;
      n = 0;
      c = 0;
      VN0_counter = 0;

      total_VN_n = 1142;
      total_CN_n = 573;
      max_VN_to_CN = 6;
      max_CN_to_VN = 7;
      min_CN_value = 2304;

      PIPELINE(pipe).flush();
    }
}

OPERATION main
{
  DECLARE
    {
      INSTANCE fetch;
    }

  BEHAVIOR
    {
      ENABLE_REG = ENABLE;
     PIPELINE(pipe).execute();
      PIPELINE(pipe).shift();
      cycle += 1;
    }

  ACTIVATION
    {
      if (!pipe.DE.IN.stalled())
            {
          fetch
        }
    }
}

OPERATION fetch IN pipe.FE
{
  DECLARE
      {
      INSTANCE decode;
      INSTANCE adquire_state, adquire_virt_data;
      }

  BEHAVIOR
    {
      short tmp_FPC=FPC;
      if(BPC_valid!=0)
            {
                  tmp_FPC=BPC;
                  BPC_valid=0;
            }
      FPC = tmp_FPC + 2;
     char tmp1, tmp2;
      progr_bus.read(tmp_FPC+1, &tmp1);
      progr_bus.read(tmp_FPC, &tmp2);
     FE.OUT.instr_reg = (((unsigned char)tmp1) << 8) | ((unsigned char)tmp2);
      FE.OUT.PC=tmp_FPC;
     instruction_counter += 1;
     adquire_virt_data();
      adquire_state();
   }
   ACTIVATION{ decode }
}

OPERATION decode IN pipe.DE
{
  DECLARE
    {
      GROUP instruction = { branch_instr || main_op };
    }
   CODING AT ( DE.IN.PC ) { DE.IN.instr_reg == instruction }
  SYNTAX { instruction }
  BEHAVIOR
   {
      EPC=DE.IN.PC;
   }
  ACTIVATION { instruction }
 }

OPERATION adquire_virt_data IN pipe.FE
{
 BEHAVIOR
      {
      bit[32] tmp_data_in;
      bit[32] tmp_reg;
      bit[12] gen_reg;
      uint32 gen_int;
      bit[12] gen_old0;
      bit[12] gen_old1;
      bit[12] gen_old2;
      bit[12] gen_old3;
      bit[12] gen_old4;
      bit[12] gen_old5;
      bit[12] gen_old6;

            if ((!virt_data0_in_ef)||(!virt_data2_in_ef)||(!virt_data3_in_ef))
                  {
                    if (!virt_data0_in_ef)
                        {
                              virt_data0_in_re = 1;
                              tmp_data_in = virt_data0_in;
                              virt_data2_in_re = 0;
                              virt_data3_in_re = 0;
                        }
                    else
                        {
                          if (!virt_data2_in_ef)
                              {
                                    virt_data2_in_re = 1;
                                    tmp_data_in = virt_data2_in;
                                    virt_data0_in_re = 0;
                                    virt_data3_in_re = 0;
                              }
                          else
                              {
                                    virt_data3_in_re = 1;
                                    tmp_data_in = virt_data3_in;
                                    virt_data0_in_re = 0;
                                    virt_data2_in_re = 0;
                              }
                        }
                        gen_reg = tmp_data_in.to_bitvec(6, 12);
                        gen_int = gen_reg.to_uint32(0, 12);

                        if (gen_int >= min_CN_value)
                          {
                        VN_to_CN_bus.read(VN_to_CN_counter +
                              max_CN_to_VN*(gen_int - min_CN_value), &tmp_reg);
                        gen_old0 = tmp_reg.to_bitvec(6, 12);

                        VN_to_CN_bus.read(VN_to_CN_counter +
                           max_CN_to_VN*(gen_int - min_CN_value) + 1, &tmp_reg);
                        gen_old1 = tmp_reg.to_bitvec(6, 12);

                        VN_to_CN_bus.read(VN_to_CN_counter +
                           max_CN_to_VN*(gen_int - min_CN_value) + 2, &tmp_reg);
                        gen_old2 = tmp_reg.to_bitvec(6, 12);

                        VN_to_CN_bus.read(VN_to_CN_counter +
                           max_CN_to_VN*(gen_int - min_CN_value) + 3, &tmp_reg);
                        gen_old3 = tmp_reg.to_bitvec(6, 12);

                        VN_to_CN_bus.read(VN_to_CN_counter +
                           max_CN_to_VN*(gen_int - min_CN_value) + 4, &tmp_reg);
                        gen_old4 = tmp_reg.to_bitvec(6, 12);

                        VN_to_CN_bus.read(VN_to_CN_counter +
                           max_CN_to_VN*(gen_int - min_CN_value) + 5, &tmp_reg);
                        gen_old5 = tmp_reg.to_bitvec(6, 12);

                        VN_to_CN_bus.read(VN_to_CN_counter +
                           max_CN_to_VN*(gen_int - min_CN_value) + 6, &tmp_reg);
                        gen_old6 = tmp_reg.to_bitvec(6, 12);

                        if (gen_old0 != gen_reg)
                                {
                        VN_to_CN_bus.write(VN_to_CN_counter +
                           max_CN_to_VN*(gen_int - min_CN_value), &tmp_data_in);
                                }
                        else
                                {
                        if (gen_old1 != gen_reg)
                                {
                        VN_to_CN_bus.write(VN_to_CN_counter +
                       max_CN_to_VN*(gen_int - min_CN_value) + 1, &tmp_data_in);
                                }
                        else
                               {
                        if (gen_old2 != gen_reg)
                                {
                        VN_to_CN_bus.write(VN_to_CN_counter +
                       max_CN_to_VN*(gen_int - min_CN_value) + 2, &tmp_data_in);
```

```
         }
 else
         {
 if (gen_old3 != gen_reg)
         {
 VN_to_CN_bus.write(VN_to_CN_counter +
max_CN_to_VN*(gen_int - min_CN_value) + 3, &tmp_data_in);
         }
 else
         {
 if (gen_old4 != gen_reg)
         {
 VN_to_CN_bus.write(VN_to_CN_counter +
max_CN_to_VN*(gen_int - min_CN_value) + 4, &tmp_data_in);
         }
 else
         {
 if (gen_old5 != gen_reg)
         {
 VN_to_CN_bus.write(VN_to_CN_counter +
max_CN_to_VN*(gen_int - min_CN_value) + 5, &tmp_data_in);
         }
 else
         {
 VN_to_CN_bus.write(VN_to_CN_counter +
max_CN_to_VN*(gen_int - min_CN_value) + 6, &tmp_data_in);
                                       }
                                 }
                           }
                     }
               }
         }
   }
 else
   {
 if (gen_reg != 0)
         {
       CN_to_VN_bus.read(CN_to_VN_counter +
                         max_VN_to_CN*gen_int, &tmp_reg);
       gen_old0 = tmp_reg.to_bitvec(6, 12);

       CN_to_VN_bus.read(CN_to_VN_counter +
                     max_VN_to_CN*gen_int + 1, &tmp_reg);
       gen_old1 = tmp_reg.to_bitvec(6, 12);

       CN_to_VN_bus.read(CN_to_VN_counter +
                     max_VN_to_CN*gen_int + 2, &tmp_reg);
       gen_old2 = tmp_reg.to_bitvec(6, 12);

       CN_to_VN_bus.read(CN_to_VN_counter +
                     max_VN_to_CN*gen_int + 3, &tmp_reg);
       gen_old3 = tmp_reg.to_bitvec(6, 12);
```

```
CN_to_VN_bus.read(CN_to_VN_counter +
              max_VN_to_CN*gen_int + 4, &tmp_reg);
gen_old4 = tmp_reg.to_bitvec(6, 12);

CN_to_VN_bus.read(CN_to_VN_counter +
              max_VN_to_CN*gen_int + 5, &tmp_reg);
gen_old5 = tmp_reg.to_bitvec(6, 12);

if (gen_old0 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
              max_VN_to_CN*gen_int, &tmp_data_in);
        }
else
        {
if (gen_old1 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
          max_VN_to_CN*gen_int + 1, &tmp_data_in);
        }
else
        {
if (gen_old2 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
          max_VN_to_CN*gen_int + 2, &tmp_data_in);
        }
else
        {
if (gen_old3 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
          max_VN_to_CN*gen_int + 3, &tmp_data_in);
        }
else
        {
if (gen_old4 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
          max_VN_to_CN*gen_int + 4, &tmp_data_in);
        }
else
        {
CN_to_VN_bus.write(CN_to_VN_counter +
          max_VN_to_CN*gen_int + 5, &tmp_data_in);
                                    }
                              }
                        }
                  }
            }
      }
```

```
                              else
                                {
                              CN_to_VN_bus.write(CN_to_VN_counter + VN0_counter,
                                                                   &tmp_data_in);
                              VN0_counter++;
                                }
                           }
                  }
         }
}

OPERATION adquire_state IN pipe.FE
{

  BEHAVIOR
         {
         STATUS_OBJ0 = new_status.to_bool(0);
         STATUS_OBJ1 = new_status.to_bool(1);
               if (STATUS_OBJ0 == 0 && STATUS_OBJ1 == 0)
                     {
                     current_status = 0;
                     }
               else
                     {
                     if (STATUS_OBJ0 == 1 && STATUS_OBJ1 == 0)
                           {
                           current_status = 1;
                           }
                     else
                           {
                           if (STATUS_OBJ0 == 0 && STATUS_OBJ1 == 1)
                                 {
                                 current_status = 2;
                                 }
                           else
                                 {
                                 current_status = 3;
                                 }
                           }
                     }
         }
}

OPERATION branch_instr IN pipe.DE
{
  DECLARE
    {
      GROUP address = { addr8 };
     GROUP insn = { BC || B };
    }
 CODING { insn address }
 SYNTAX { insn }
 BEHAVIOR
    {
      AR = ((R[15] << 8) | address);
     pipe.DE.IN.stall();
    }
 ACTIVATION { insn }
}

OPERATION main_op IN pipe.DE
{
  DECLARE
      {
        GROUP instr = { SET || RESET || send_VN || create_CN || create_VN ||
                                       send_CN || send_output || charge_input };
      }
 CODING { instr }
 SYNTAX { instr }
 ACTIVATION { instr }
}
```

9 Appendix B (instructions.lisa)

```
OPERATION BC IN pipe.EX
{
  DECLARE
    {
      GROUP condition = { EQ || NEQ };
      REFERENCE address;
      INSTANCE ctrl_reg;
    }
 CODING { 0b1000 condition ctrl_reg }
  SYNTAX { "BC" ~" " address "," ctrl_reg ~":" ~condition }
 SWITCH (condition) {
    CASE EQ: {
      BEHAVIOR
            {
            virt_data0_out_we = 0;
            virt_data2_out_we = 0;
            virt_data3_out_we = 0;
            if (ctrl_reg == 0)
              {
                  BPC = AR;
                  BPC_valid = 1;
                  pipe.DE.IN.flush();
              }
            }
    }
    CASE NEQ: {
      BEHAVIOR
            {
            virt_data0_out_we = 0;
            virt_data2_out_we = 0;
            virt_data3_out_we = 0;
           if (ctrl_reg == 1)
             {
                  BPC = AR;
                  BPC_valid = 1;
                  pipe.DE.IN.flush();
             }
      }
    }
  }
}
 OPERATION B IN pipe.EX
{
   DECLARE
    {
      REFERENCE address;
    }
 CODING { 0b01000001 }
  SYNTAX { "B" ~" " address }
  BEHAVIOR
    {
      virt_data0_out_we = 0;
      virt_data2_out_we = 0;
      virt_data3_out_we = 0;
      BPC = AR;
      BPC_valid = 1;
      pipe.DE.IN.flush();
    }
}

OPERATION RESET IN pipe.EX
{
  DECLARE
      {
        INSTANCE ctrl_reg;
      }
 CODING { 0b01000010 ctrl_reg 0bx[5]}
  SYNTAX { "RESET" ~" " ctrl_reg }
 BEHAVIOR
      {
      ctrl_reg = 0;
    }
}

OPERATION SET IN pipe.EX
{
  DECLARE
      {
        INSTANCE ctrl_reg;
      }
 CODING { 0b01000110 ctrl_reg 0bx[5]}
  SYNTAX { "SET" ~" " ctrl_reg }
 BEHAVIOR
      {
      ctrl_reg = 1;
    }
}

OPERATION send_VN IN pipe.EX
{
  CODING { 0b01000011 0bx[8] }
  SYNTAX { "send_VN" }
 BEHAVIOR
      {
        bit[32] tmp_data_out;
        bit[32] tmp_reg;
        bit[1] itf_to_send_bit0;
        bit[1] itf_to_send_bit1;
        bit[12] gen_reg;
```

```
uint32 gen_int;
bit[12] gen_old0;
bit[12] gen_old1;
bit[12] gen_old2;
bit[12] gen_old3;
bit[12] gen_old4;
bit[12] gen_old5;
bit[12] gen_old6;
VN0_counter = 0;

if (VN_temp_count < total_VN_n)
    {
    if (n < max_VN_to_CN)
          {
          VNi_bus.read(VNi_counter, &tmp_data_out);
          gen_reg = tmp_data_out.to_bitvec(6, 12);
          gen_int = gen_reg.to_uint32(0, 12);
          VNi_counter++;
          n++;
          if (gen_int != 0)
                {
                itf_to_send_bit0 = tmp_data_out.to_bitvec(30, 1);
                itf_to_send_bit1 = tmp_data_out.to_bitvec(31, 1);

          if (!itf_to_send_bit0)
                {
          if (!itf_to_send_bit1)
                {
                virt_data0_out_we = 0;
                virt_data2_out_we = 0;
                virt_data3_out_we = 0;

                VN_to_CN_bus.read(VN_to_CN_counter +
                       max_CN_to_VN*(gen_int - min_CN_value), &tmp_reg);
                gen_old0 = tmp_reg.to_bitvec(6, 12);

                VN_to_CN_bus.read(VN_to_CN_counter +
                   max_CN_to_VN*(gen_int - min_CN_value) + 1, &tmp_reg);
                gen_old1 = tmp_reg.to_bitvec(6, 12);

                VN_to_CN_bus.read(VN_to_CN_counter +
                   max_CN_to_VN*(gen_int - min_CN_value) + 2, &tmp_reg);
                gen_old2 = tmp_reg.to_bitvec(6, 12);

                VN_to_CN_bus.read(VN_to_CN_counter +
                   max_CN_to_VN*(gen_int - min_CN_value) + 3, &tmp_reg);
                gen_old3 = tmp_reg.to_bitvec(6, 12);

                VN_to_CN_bus.read(VN_to_CN_counter +
                   max_CN_to_VN*(gen_int - min_CN_value) + 4, &tmp_reg);
                gen_old4 = tmp_reg.to_bitvec(6, 12);
```

```
      VN_to_CN_bus.read(VN_to_CN_counter +
         max_CN_to_VN*(gen_int - min_CN_value) + 5, &tmp_reg);
      gen_old5 = tmp_reg.to_bitvec(6, 12);

      VN_to_CN_bus.read(VN_to_CN_counter +
         max_CN_to_VN*(gen_int - min_CN_value) + 6, &tmp_reg);
      gen_old6 = tmp_reg.to_bitvec(6, 12);

if (gen_old0 != gen_reg)
      {
      VN_to_CN_bus.write(VN_to_CN_counter +
        max_CN_to_VN*(gen_int - min_CN_value), &tmp_data_out);
      }
else
      {
if (gen_old1 != gen_reg)
      {
      VN_to_CN_bus.write(VN_to_CN_counter +
    max_CN_to_VN*(gen_int - min_CN_value) + 1, &tmp_data_out);
      }
else
      {
if (gen_old2 != gen_reg)
      {
      VN_to_CN_bus.write(VN_to_CN_counter +
    max_CN_to_VN*(gen_int - min_CN_value) + 2, &tmp_data_out);
      }
else
      {
if (gen_old3 != gen_reg)
      {
      VN_to_CN_bus.write(VN_to_CN_counter +
    max_CN_to_VN*(gen_int - min_CN_value) + 3, &tmp_data_out);
      }
else
      {
if (gen_old4 != gen_reg)
      {
      VN_to_CN_bus.write(VN_to_CN_counter +
    max_CN_to_VN*(gen_int - min_CN_value) + 4, &tmp_data_out);
      }
else
      {
if (gen_old5 != gen_reg)
      {
      VN_to_CN_bus.write(VN_to_CN_counter +
    max_CN_to_VN*(gen_int - min_CN_value) + 5, &tmp_data_out);
      }
else
      {
      VN_to_CN_bus.write(VN_to_CN_counter +
    max_CN_to_VN*(gen_int - min_CN_value) + 6, &tmp_data_out);
```

```
      }
                                       }
                                 }
                           }
                     }
              }
      }
else
      {
      virt_data3_out_we = 0;
      virt_data0_out_we = 0;
      if (!virt_data2_out_ff)
            {
            virt_data2_out_we = 1;
            virt_data2_out = tmp_data_out;
            }
            else
            {
            virt_data2_out_we = 0;
            }
      }
}
else
{
      if (!itf_to_send_bit1)
            {
            virt_data3_out_we = 0;
            virt_data2_out_we = 0;
            if (!virt_data0_out_ff)
                  {
                  virt_data0_out_we = 1;
                  virt_data0_out = tmp_data_out;
                  }
            else
                  {
                  virt_data0_out_we = 0;
                  }
            }
            else
                  {
                  virt_data0_out_we = 0;
                  virt_data2_out_we = 0;
                  if (!virt_data3_out_ff)
                        {
                        virt_data3_out_we = 1;
                        virt_data3_out = tmp_data_out;
                        }
                  else
                        {
                        virt_data3_out_we = 0;
                        }
                  }
```

```
                              }
                        }
                     else
                          {
                              virt_data0_out_we = 0;
                              virt_data2_out_we = 0;
                              virt_data3_out_we = 0;
                          }
                  }
                else
                  {
                      virt_data0_out_we = 0;
                      virt_data2_out_we = 0;
                      virt_data3_out_we = 0;
                      n = 0;
                      VN_temp_count++;
                  }
            }
          else
            {
                DONE = 1;
                VNi_counter = 0x1000;
                VN_temp_count = 0;
                n = 0;
                virt_data0_out_we = 0;
                virt_data2_out_we = 0;
                virt_data3_out_we = 0;
            }
    }
}

OPERATION create_CN IN pipe.EX
{
  CODING { 0b01000100 0bx[8] }
  SYNTAX { "create_CN" }
 BEHAVIOR
      {
        uint8 p;
        uint8 g;
        uint8 z;
        bit[32] zero_reg;
        bit[12] temp;
        bit[32] tmp_reg;
        bit[32] tmp_reg2;
        bit[8] data_out;
        bit[12] gen_reg;
        uint32 gen_int;
        VN0_counter = 0;

        if (CN_temp_count < total_CN_n)
            {
              zero_reg = 0x00000000;
```

```
if (s < (max_CN_to_VN + 1))
    {
      if (s == 0)
          {
            CNi_bus.read(CNi_counter, &tmp_reg2);
            temp = tmp_reg2.to_bitvec(18, 12);
            counter = temp.to_uint32(0, 12);
            s++;
          }
      else
          {
            CNi_bus.read(CNi_counter + s - 1, &tmp_reg2);
            gen_reg = tmp_reg2.to_bitvec(18, 12);
            gen_int = gen_reg.to_uint32(0, 12);
            if (gen_int == counter)
                {
                  reg_home_calc[m] = tmp_reg2.to_bitvec(6, 12);
                  m++;
                  VN_to_CN_bus.read(VN_to_CN_counter +
         max_CN_to_VN*(counter - min_CN_value) + s - 1, &tmp_reg);
                  temp_calc[n] = tmp_reg.to_bitvec(0, 6);
                  reg_calc[n] = tmp_reg.to_bitvec(18, 12);
                  n++;
                }
          }
      s++;
    }
else
    {
      if (c < m)
          {
            data_out = 0xff;
            for (p = 0 ; p < n ; p++)
                {
                  if (reg_calc[p] == reg_home_calc[c])
                      {
                        z = p;
                      }
                }
            for (g = 0 ; g < n ; g++)
                {
                  if (g != z)
                      {
                        if ((temp_calc[g] & 0x3f)<(data_out&0x3f))
                                  {
                                    data_out = (temp_calc[g]&0x3f);
                                  }
                            }
                }
            data_out = data_out & 0x3f;
            CNi_bus.write(CNi_counter + c, &data_out, 1, 0, 6);
            c++;
```

```
                        }
                    else
                        {
                          VN_to_CN_bus.write(VN_to_CN_counter +
                              max_CN_to_VN*(counter - min_CN_value), &zero_reg);
                          VN_to_CN_bus.write(VN_to_CN_counter +
                          max_CN_to_VN*(counter - min_CN_value) + 1, &zero_reg);
                          VN_to_CN_bus.write(VN_to_CN_counter +
                          max_CN_to_VN*(counter - min_CN_value) + 2, &zero_reg);
                          VN_to_CN_bus.write(VN_to_CN_counter +
                          max_CN_to_VN*(counter - min_CN_value) + 3, &zero_reg);
                          VN_to_CN_bus.write(VN_to_CN_counter +
                          max_CN_to_VN*(counter - min_CN_value) + 4, &zero_reg);
                          VN_to_CN_bus.write(VN_to_CN_counter +
                          max_CN_to_VN*(counter - min_CN_value) + 5, &zero_reg);
                          VN_to_CN_bus.write(VN_to_CN_counter +
                          max_CN_to_VN*(counter - min_CN_value) + 6, &zero_reg);
                          CN_temp_count++;
                          CNi_counter = CNi_counter + max_CN_to_VN;
                          n = 0;
                          z = 0;
                          s = 0;
                          p = 0;
                          c = 0;
                          m = 0;
                          g = 0;
                        }
                  }
            }
        else
            {
              CN_temp_count = 0;
              CNi_counter = 0x3000;
              VN_to_CN_counter = 0x5000;
              DONE = 1;
              n = 0;
              z = 0;
              s = 0;
              p = 0;
              c = 0;
              m = 0;
              g = 0;
            }
      }
}

OPERATION send_CN IN pipe.EX
{
  CODING { 0b01000101 0bx[8] }
  SYNTAX { "send_CN" }
 BEHAVIOR
      {
```

```
bit[32] tmp_data_out;
bit[32] tmp_reg;
bit[1] itf_to_send_bit0;
bit[1] itf_to_send_bit1;
bit[12] gen_reg;
uint32 gen_int;
bit[12] ctrl_reg;
uint32 ctrl_int;
bit[12] gen_old0;
bit[12] gen_old1;
bit[12] gen_old2;
bit[12] gen_old3;
bit[12] gen_old4;
bit[12] gen_old5;

if (CN_temp_count < total_CN_n)
    {
      if (n < max_CN_to_VN)
          {
            CNi_bus.read(CNi_counter, &tmp_data_out);
            gen_reg = tmp_data_out.to_bitvec(6, 12);
            gen_int = gen_reg.to_uint32(0, 12);
            ctrl_reg = tmp_data_out.to_bitvec(18, 12);
            ctrl_int = ctrl_reg.to_uint32(0, 12);
            CNi_counter++;
            n++;

            if (ctrl_int != 0)
                {
                  itf_to_send_bit0 = tmp_data_out.to_bitvec(30, 1);
                  itf_to_send_bit1 = tmp_data_out.to_bitvec(31, 1);

                  if (!itf_to_send_bit0)
                    {
                  if (!itf_to_send_bit1)
                    {
                      virt_data0_out_we = 0;
                      virt_data2_out_we = 0;
                      virt_data3_out_we = 0;
                      if (gen_reg != 0)
                            {
                      CN_to_VN_bus.read(CN_to_VN_counter +
                                        max_VN_to_CN*gen_int, &tmp_reg);
                      gen_old0 = tmp_reg.to_bitvec(6, 12);

                      CN_to_VN_bus.read(CN_to_VN_counter +
                                    max_VN_to_CN*gen_int + 1, &tmp_reg);
                      gen_old1 = tmp_reg.to_bitvec(6, 12);

                      CN_to_VN_bus.read(CN_to_VN_counter +
                                    max_VN_to_CN*gen_int + 2, &tmp_reg);
                      gen_old2 = tmp_reg.to_bitvec(6, 12);
```

```
CN_to_VN_bus.read(CN_to_VN_counter +
              max_VN_to_CN*gen_int + 3, &tmp_reg);
gen_old3 = tmp_reg.to_bitvec(6, 12);

CN_to_VN_bus.read(CN_to_VN_counter +
              max_VN_to_CN*gen_int + 4, &tmp_reg);
gen_old4 = tmp_reg.to_bitvec(6, 12);

CN_to_VN_bus.read(CN_to_VN_counter +
              max_VN_to_CN*gen_int + 5, &tmp_reg);
gen_old5 = tmp_reg.to_bitvec(6, 12);

if (gen_old0 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
             max_VN_to_CN*gen_int, &tmp_data_out);
        }
else
        {
if (gen_old1 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
         max_VN_to_CN*gen_int + 1, &tmp_data_out);
        }
else
        {
if (gen_old2 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
         max_VN_to_CN*gen_int + 2, &tmp_data_out);
        }
else
        {
if (gen_old3 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
         max_VN_to_CN*gen_int + 3, &tmp_data_out);
        }
else
        {
if (gen_old4 != gen_reg)
        {
CN_to_VN_bus.write(CN_to_VN_counter +
         max_VN_to_CN*gen_int + 4, &tmp_data_out);
        }
else
        {
CN_to_VN_bus.write(CN_to_VN_counter +
         max_VN_to_CN*gen_int + 5, &tmp_data_out);
                                }
                          }
```

```
                        }
                  }
            }
        }
    else
          {
    CN_to_VN_bus.write(CN_to_VN_counter + VN0_counter,
                                        &tmp_data_out);
    VN0_counter++;
                {
          }
     else
        {
            virt_data3_out_we = 0;
            virt_data0_out_we = 0;
            if (!virt_data2_out_ff)
              {
                  virt_data2_out_we = 1;
                  virt_data2_out = tmp_data_out;
              }
            else
              {
                  virt_data2_out_we = 0;
              }
        }
  }
else
  {
      if (!itf_to_send_bit1)
        {
            virt_data3_out_we = 0;
            virt_data2_out_we = 0;
            if (!virt_data0_out_ff)
              {
                  virt_data0_out_we = 1;
                  virt_data0_out = tmp_data_out;
              }
            else
              {
                  virt_data0_out_we = 0;
              }
        }
      else
        {
                virt_data0_out_we = 0;
                virt_data2_out_we = 0;
                if (!virt_data3_out_ff)
                  {
                      virt_data3_out_we = 1;
                      virt_data3_out = tmp_data_out;
                  }
                else
```

```
                                            {
                                                virt_data3_out_we = 0;
                                            }
                                    }
                              }
                            }
                        else
                            {
                              virt_data0_out_we = 0;
                              virt_data2_out_we = 0;
                              virt_data3_out_we = 0;
                            }
                        }
                      else
                        {
                            virt_data0_out_we = 0;
                            virt_data2_out_we = 0;
                            virt_data3_out_we = 0;
                            n = 0;
                            CN_temp_count++;
                        }
                  }
                else
                  {
                      DONE = 1;
                      CNi_counter = 0x3000;
                      CN_temp_count = 0;
                      n = 0;
                      virt_data0_out_we = 0;
                      virt_data2_out_we = 0;
                      virt_data3_out_we = 0;
                  }

        }
}

OPERATION create_VN IN pipe.EX
{
  CODING { 0b01000111 0bx[8] }
  SYNTAX { "create_VN" }
 BEHAVIOR
      {
        uint8 p;
        uint8 g;
        uint8 z;
        bit[32] zero_reg;
        bit[12] temp;
        bit[32] tmp_reg;
        bit[32] tmp_reg2;
        bit[8] data_to_output;
        bit[8] data_out;
        bit[12] gen_reg;
```

```
uint32 gen_int;
VN0_counter = 0;

if (VN_temp_count < total_VN_n)
    {
      zero_reg = 0x00000000;
      if (s < (max_VN_to_CN + 1))
          {
            if (s == 0)
                {
                  VNi_bus.read(VNi_counter, &tmp_reg2);
                  temp = tmp_reg2.to_bitvec(18, 12);
                  counter = temp.to_uint32(0, 12);
                  s++;
                }
            else
                {
                  VNi_bus.read(VNi_counter + s - 1, &tmp_reg2);
                  gen_reg = tmp_reg2.to_bitvec(18, 12);
                  gen_int = gen_reg.to_uint32(0, 12);
                  if (gen_int == counter)
                      {
                        reg_home_calc[m] = tmp_reg2.to_bitvec(6, 12);
                        m++;
                        CN_to_VN_bus.read(CN_to_VN_counter +
                                max_VN_to_CN*counter + s - 1, &tmp_reg);
                        temp_calc[n] = tmp_reg.to_bitvec(0, 6);
                        reg_calc[n] = tmp_reg.to_bitvec(18, 12);
                        n++;
                      }
                }
            s++;
          }
      else
          {
            if (c < m)
                {
                  input_bus.read(input_counter, &data_to_output);
                  data_to_output = data_to_output & 0x3f;
                  data_out = data_to_output;
                  for (p = 0 ; p < n ; p++)
                      {
                        if (reg_calc[p] == reg_home_calc[c])
                            {
                              z = p;
                            }
                      }
                 for (g = 0 ; g < n ; g++)
                      {
                      data_to_output=data_to_output+(temp_calc[g]&0x3f);
                        if (g != z)
                            {
                              data_out = data_out + (temp_calc[g]&0x3f);
                            }
                      }
                  data_out = data_out & 0x3f;
                  data_to_output = data_to_output & 0x3f;
                  VNi_bus.write(VNi_counter + c, &data_out, 1, 0, 6);
                  output_bus.write(output_counter, &data_to_output);
                  c++;
                }
            else
                {
                  CN_to_VN_bus.write(CN_to_VN_counter +
                                        max_VN_to_CN*counter, &zero_reg);
                  CN_to_VN_bus.write(CN_to_VN_counter +
                                   max_VN_to_CN*counter + 1, &zero_reg);
                  CN_to_VN_bus.write(CN_to_VN_counter +
                                   max_VN_to_CN*counter + 2, &zero_reg);
                  CN_to_VN_bus.write(CN_to_VN_counter +
                                   max_VN_to_CN*counter + 3, &zero_reg);
                  CN_to_VN_bus.write(CN_to_VN_counter +
                                   max_VN_to_CN*counter + 4, &zero_reg);
                  CN_to_VN_bus.write(CN_to_VN_counter +
                                   max_VN_to_CN*counter + 5, &zero_reg);
                  VN_temp_count++;
                  input_counter++;
                  output_counter++;
                  VNi_counter = VNi_counter + max_VN_to_CN;
                  n = 0;
                  z = 0;
                  s = 0;
                  p = 0;
                  c = 0;
                  m = 0;
                  g = 0;
                }
          }
    }
else
    {
      VN_temp_count = 0;
      input_counter = 0x9000;
      output_counter = 0xa000;
      VNi_counter = 0x1000;
      DONE = 1;
      n = 0;
      z = 0;
      s = 0;
      p = 0;
      c = 0;
      m = 0;
      g = 0;
    }
}
}

OPERATION send_output IN pipe.EX
{
CODING { 0b01001000 0bx[8] }
SYNTAX { "send_output" }
BEHAVIOR
{
  bit[32] temporary_data;
  bit[6] tmp_output;
  bool bit_out;

  if (VN_temp_count < total_VN_n)
      {
      if (n < 32)
          {
                output_bus.read(output_counter + VN_temp_count, &tmp_output);
                bit_out = tmp_output.to_bool(5);
                temporary_data.assign(bit_out, n, 1);
                tmp_data = temporary_data;
                virt_data1_out_we = 0;
                VN_temp_count++;
                n++;
          }
      else
          {
                if (!virt_data1_out_ff)
                  {
                      virt_data1_out = tmp_data;
                      virt_data1_out_we = 1;
                      n = 0;
                  }
                else
                  {
                      virt_data1_out_we = 0;
                  }
          }
      }
  else
      {
          if (n != 0)
                {
                  temporary_data.assign(0, n, 32-n);
                  tmp_data = temporary_data;
                  if (!virt_data1_out_ff)
                    {
                        virt_data1_out = tmp_data;
                        virt_data1_out_we = 1;
                        n = 0;
                    }
                  else
```

```
                        {
                            virt_data1_out_we = 0;
                        }
                    }
              else
                    {
                    DONE = 1;
                    virt_data1_out_we = 0;
                    }
            }
    }
}

OPERATION charge_input IN pipe.EX
{
 CODING { 0b01001001 0bx[8] }
  SYNTAX { "charge_input" }
 BEHAVIOR
      {
            bit[32] tmp_data_in;
            bit[8] data_in;

            if (VN_temp_count < total_VN_n)
              {
                    VNi_bus.read(VNi_counter, &tmp_data_in);
                    data_in = tmp_data_in.to_bitvec(0, 8) & 0x3f;
                    input_bus.write(input_counter, &data_in);
                    VNi_counter = VNi_counter + max_VN_to_CN;
                    input_counter++;
                    VN_temp_count++;
              }
            else
              {
                    DONE = 1;
                    VNi_counter = 0x1000;
                    input_counter = 0x9000;
                    VN_temp_count = 0;
              }
      }
}
```

10 Appendix C (progr_main.asm)

```
;=================================================================
; FILE:        progr_main.asm
; DESCRIPTION: main ASIP program
;=================================================================
;================================
;filling data memory VNi_mem

      .section .data_VN, "aw", @progbits
      .word 0x0016921f
      .word 0x0017165f
      .word 0x0017569f
      .word 0x00000000
      .word 0x00000000
      .word 0x00000000
      .word 0x001a9260
      .word 0x001b16a0
      .word 0x001b56e0
      .word 0x00000000
      .word 0x00000000
      .word 0x00000000
….................................
;================================
;filling data memory CNi_mem

      .section .data_CN, "aw", @progbits
      .word 0x24081800
      .word 0x240842c0
      .word 0x2408ce40
      .word 0x2408ed40
      .word 0x24092240
      .word 0x24093880
      .word 0x00000000
      .word 0x241418c0
      .word 0x64144380
      .word 0x2414cf00
      .word 0x2414ee00
      .word 0x24152300
      .word 0x24153940
      .word 0x00000000
….................................

                  .text
_loop:            BC @_loop, STATUS_OBJ0:EQ
                  BC @_init, STATUS_OBJ1:NEQ
_run:             BC @_run, ENABLE_REG:EQ
_wait1:           BC @_first_step, STATUS_OBJ0:NEQ
```

```
                  BC @_wait1, STATUS_OBJ1:NEQ
                  B @_out_step
_first_step:      send_VN
                  BC @_first_step, DONE:EQ
                  RESET DONE

_run_CN:          BC @_run_CN, ENABLE_REG:EQ
_wait2:           BC @_CN_step, STATUS_OBJ0:NEQ
                  BC @_wait2, STATUS_OBJ1:NEQ
                  B @_out_step
_CN_step:         create_CN
                  BC @_CN_step, DONE:EQ
                  RESET DONE
_CN_send:         send_CN
                  BC @_CN_send, DONE:EQ
                  RESET DONE

_run_VN:          BC @_run_VN, ENABLE_REG:EQ
_wait3:           BC @_VN_step, STATUS_OBJ0:NEQ
                  BC @_wait3, STATUS_OBJ1:NEQ
                  B @_out_step
_VN_step:         create_VN
                  BC @_VN_step, DONE:EQ
                  RESET DONE
_VN_send:         send_VN
                  BC @_VN_send, DONE:EQ
                  RESET DONE
                  B @_run_CN

_init:            charge_input
                  BC @_init, DONE:EQ
                  RESET DONE
                  B @_loop

_out_step:        send_output
                  BC @_out_step, DONE:EQ
                  RESET DONE
_inf_loop:        B @_inf_loop
                  .end
```

11 Appendix D (progr_main.cmd)

```
MEMORY
{
    progr_mem    : origin = 0x0000 , length = 0x1000 , bytes = 1
    VNi_mem      : origin = 0x1000 , length = 0x2000 , bytes = 4
    CNi_mem      : origin = 0x3000 , length = 0x2000 , bytes = 4
    VN_to_CN_mem : origin = 0x5000 , length = 0x2000 , bytes = 4
    CN_to_VN_mem : origin = 0x7000 , length = 0x2000 , bytes = 4
    input_mem    : origin = 0x9000 , length = 0x1000 , bytes = 1
    ouput_mem    : origin = 0xa000 , length = 0x1000 , bytes = 1
}

SECTIONS
{
    .text:    > progr_mem
    .data_VN: > VNi_mem
    .data_CN: > CNi_mem
}
```

12 Bibliography

[1] Wen Ji, Yuta Abe, Takeshi Ikenaga, Satoshi Goto. A High Performance Partially- Parallel Irregular LDPC Decoder Based on Sum-Delta Message Passing Schedule. In IEICE TRANS. FUNDAMENTALS, vol.E91-A, no.12, pp.36223629. December 2008.

[2] Jian Suan. An introduction to Low PArity Check (LDPC) Codes. WCRL Seminar Series, West Virginia University. June 3, 2003

[3] D. Declercq and F. Verdier. Optimization of LDPC Finite Precision Belief Propagation Decoding with Discrete Density Evolution. ETIS ENSEA/UCP/CNRS UMR 8051, France.

[4] Xilinx. Virtex-5 FPGA Documentation. DS100 (v5.0). www.xilinx.com. February 6, 2009.

[5] Fabrizio Vacca, Guido Masera and Hazem Moussa, Amer Baghdadi, Michel Jezequel. Flexible Architectures for LDPC Decoders based on Network On Chip Paradigm. Dipartimento di Elettronica Politecnico di Torino, Torino, Italy and Electronics Department Technopole Brest-Iroise, Bretagne, France.

[6] Federico Quaglio, Fabrizio Vacca, Cristiano Castellano, Alberto Tarable, Guido Masera. Interconnection Framework for High-Throughput, Flexible LDPC Decoders. 3-9810801-0-6/DATE06 2006 EDAA. CERCOM-Dipartimento di Elettronica Politecnico di Torino, Torino, Italy.

[7] Bernhard M.J. Leiner. LDPC Codes - a brief Tutorial. April 8, 2008.

[8] Federico Quaglio, Fabrizio Vacca, Guido Masera. Low Complexity, Flexible LDPC Decoders. Dipartimento di Elettronica Politecnico di Torino, Torino, Italy.

[9] Jean Nguyen, Dr.Borivoje Nikolic, Engling Yeo. Design of a Low Density Parity Check Iterative Decoder. University of Wisconsin, Madison and University of California, Berkeley.

[10] Ioannis Dagres, Andreas Zalonis, Nikos Dimitriou, Konstantinos Nikitopoulos, Andreas Polydoros. Flexible Radio: A Framework for Optimized Multimodal Operation via Dynamic Signal Design. EURASIP Journal on Wireless Communications and Networking 2005:3, 284-297.

[11] Jinghu Chen, Ajay Dholakia, Evangelos Eleftheriou, Marc P.C. Fossorier, Xiao-Yu Hu. Reduced-Complexity Decoding of LDPC Codes. IEEE TRANSACTIONS ON COMMUNICATIONS, vol. 53, no. 8, pp.1288-1299. August 2005.

[12] Ismael Gomez Miguelez. A Software Framework for Software Radio. Final Project. UPC, Barcelona, Spain. January, 2008.

[13] TurboBest, BEST IP FEC CORES. IEEE 802.16e LDPC Encoder/Decoder Core. www.turbobest.com.

[14] T. Theocharides, G. Link, N. Vijaykrisham, M.J. Irwin. Implementing LDPC Decoding on

NetworkOnChip. In Proc. 18th INT. CONF. ON VLSI DESIGN (VLSID05), pp. 134137. 2005.

[15] T. Brack, M. Alles, T. Lehnigk-Emden, F. Kienle, N. Wehn, N. L'Insalata, F. Rossi, M. Rovini, L. Fanucci. Low complexity ldpc decoders for next generation standards. Proc. of Design, AUTOMATION & TEST IN EUROPE CONFERENCE & EXHIBITION 2007, pp. 16. April, 2007.

[16] S. Xin-Yu, Z. Cheng-Zhou, L. Cheng-Hung, W. An-Yeu. An 8.29 mm2 52 mW multi-mode ldpc decoder design for mobile wimax system in 0.13 µm cmos process. IEEE Journal of Solid-State Circuits, vol. 43, no. 3, pp. 672683. March 2008.

[17] L. Chih-Hao, Y. Shau-Wei, C. Chih-Lung, C. Hsie-Chia, L. Chen-Yi, H. Yar-Sun, J. Shyh-Jye. An ldpd decoder chip based on self-routing network for 802.16e applications. IEEE Journal of Solid-State Circuits, vol. 43, no. 3, pp. 684694. March 2008.