University of Alberta

AN ARCHITECTURE FOR LOW-DENSITY PARITY-CHECK CONVOLUTIONAL CODE ENCODERS AND DECODERS

by

Ramkrishna Swamy

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**

in

Communications

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2007

Your file Votre référence
ISBN: 978-0-494-33356-3
Our file Notre référence
ISBN: 978-0-494-33356-3

"One has to watch out for engineers –
they begin with the sewing machine and end up with the atomic bomb."
– Marcel Pagnol

Abstract

Low-density parity-check block codes (LDPC-BCs) are quickly becoming the forward error correcting code of choice for emerging communication standards. However, low-density parity-check convolutional codes (LDPC-CCs), the convolutional counterpart of LDPC-BCs, seem to be better suited in applications with streaming data or variable sized packets. A rate-1/2, (128,3,6) LDPC-CC ASIC has been implemented in 180-nm, 1.8-V CMOS technology. We present the VLSI architecture of a register-based LDPC-CC encoder and decoder that includes an on-chip, pseudorandom additive white Gaussian noise channel emulator. The decoder comprises a pipeline of ten identical processing units and attains up to 175 Mbps of decoded throughput.

To my beloved appa, amma and akka.

Acknowledgements

The work herein was funded by the Semiconductor Research Corporation (SRC), Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Alberta.

First off, my sincere thanks to my supervisor, Dr. Stephen Bates, for his invaluable patience, guidance and continuous support during the research and writing phases. His drive to success and easy-going mentality enabled me to conquer challenges in a variety of areas including design, test, critical analysis and technical writing. Only with his support was I able to participate in several international conferences and attain a strong publication record.

Many special thanks go to Dr. Duncan Elliott, Dr. Bruce Cockburn, Dr. Vincent Gaudet and Dr. Christian Schlegel for their thought-provoking conversations and meticulous corrections regarding publications, among all else.

I wish to thank my fellow colleagues for their assistance and feedback at times of need: Zhengang Chen, John Koob, Maziyar Khorasani, Saeed Fouladi Fard, Leendert van den Berg, Dr. Amirhossein Alimohammad, Dr. Dmitri Truhachev and Dr. Sheryl Howard. An extended thank you goes to Tyler Brandon for his astute judgment calls and VLSI expertise throughout the course of my degree.

Also, thanks to Dr. Chris Winstead of Utah State University for taking the time to share and elaborate ideas on other novel circuit implementations for low-density parity-check convolutional codes.

Lastly but not least, I am very grateful to my parents and my elder sister for their invisible sacrifices and for their love, encouragement and support on countless off-days.

Table of Contents

List of Tables

List of Figures

Nomenclature

List of Acronyms

10GBASE-T	10 Gbps Ethernet over unshielded twisted pair, page 21
APP	*A Posteriori* Probability, page 18
ASIC	Application Specific Integrated Circuit, page 7
AWGN	Additive White Gaussian Noise, page 10
BER	Bit Error Rate, page 11
BF	Bit-Flipping, page 18
BIST	Built-In Self Test, page 43
BP	Belief Propagation, page 18
BPSK	Binary Phase Shift Keying, page 11
CD	Compact Disc, page 4
CLT	Central Limit Theorem, page 44
CMC	Canadian Microelectronics Corporation, page 65
CMOS	Complementary Metal-Oxide Semiconductor, page 36
CQFP	Ceramic Quad Flat-Pack, page 57
dB	Decibel, page 9
DRC	Design Rule Check, page 56
DUT	Device Under Test, page 65
DVB-S2	Digital Video Broadcasting Satellite, page 79
DVD	Digital Versatile Disc, page 1
ECC	Error-Correcting Code, page 4

EDA	Electronic Design Automation, page 56
FEC	Forward Error Correction, page 1
FER	Frame Error Rate, page 11
FIFO	First-In First-Out, page 28
FPGA	Field Programmable Gate Array, page 43
FSM	Finite State Machine, page 63
Gbps	Gigabits Per Second, page 21
GF(·)	Galois Field, page 4
I/O	Input/Output, page 56
IEEE	Institute of Electrical and Electronics Engineers, page 5
JPEG	Joint Photographic Experts Group, page 4
LDPC	Low-Density Parity-Check, page 1
LDPC-BC	Low-Density Parity-Check Block Code, page 13
LDPC-CC	Low-Density Parity-Check Convolutional Code, page 13
LFSR	Linear Feedback Shift Register, page 44
LLR	Log-Likelihood Ratio, page 18
LSB	Least Significant Bit, page 6
LVS	Layout-Versus-Schematic, page 56
LZW	Lempel-Ziv-Welch, page 4
MAC	Media (or Medium) Access Control, page 5
Mbps	Megabits Per Second, page 7
MLG	Majority-Logic, page 18
MPA	Message-Passing Algorithm, page 18
MSB	Most Significant Bit, page 6
OFDMA	Orthogonal Frequency Division Multiple Access, page 7
OSI	Open Systems Interconnection, page 5

PC	Personal Computer, page 65
PCB	Printed Circuit Board, page 65
PCN	Parity-Check Node, page 16
PDF	Portable Document Format, page 4
PHY	**Phy**sical layer, page 5
PN	Pseudorandom Noise, page 6
PSK	Phase Shift Keying, page 11
QAM	Quadrature Amplitude Modulation, page 7
QPSK	Quadrature Phase Shift Keying, page 7
RAM	Random Access Memory, page 63
ROM	Read Only Memory, page 62
RTL	Register-Transfer Level, page 56
SNR	Signal-to-Noise Ratio, page 2
SPA	Sum-Product Algorithm, page 18
TSMC	Taiwan Semiconductor Manufacturing Corporation, page 43
VHDL	Very-high-speed-integrated-circuit Hardware Description Language, page 43
VLSI	Very Large Scale Integration, page 41
VN	Variable Node, page 16
Wi-LAN	**Wi**reless **L**ocal **A**rea **N**etwork, page 21
WiMAX	**W**orldwide **I**nteroperability for **M**icrowave **Ac**cess, page 5
XOR	Exclusive-OR, page 18

List of Symbols

$\boldsymbol{G}$	Generator matrix of linear FEC codes, page 14
$\boldsymbol{H}$	Parity-check matrix of linear FEC codes, page 14
$\boldsymbol{r}$	Received vector or block at the receiver, page 14
$\boldsymbol{u}$	Input message vector or block, page 14
$\boldsymbol{u}(t)$	Time-varying information bitstream in LDPC-CCs, page 22
$\boldsymbol{v}$	Codeword vector or block, page 14
$\boldsymbol{v}(t)$	Time-varying, systematically encoded bitstream in LDPC-CCs, i.e. source bits are included in the encoded bitstream, page 22
ℓ	Denotes a log-likelihood ratio, page 18
$\ell_u(t)$	LLR of an information-bit message in LDPC-CCs, page 29
$\ell_v(t)$	LLR of an code-bit message in LDPC-CCs, page 29
$\hat{\boldsymbol{s}}$	Syndrome vector, page 14
$\hat{u}(t)$	Decoded (reconstructed), time-varying information bitstream in LDPC-CCs, page 28
μ	Mean in a Gaussian probability distribution, page 10
$\phi(t)$	Denotes the phase relationship in the semi-infinite $\boldsymbol{H}^T$ matrix for LDPC-CCs, page 25
σ	Standard deviation in a Gaussian probability distribution, page 10
σ^2	Variance in a Gaussian probability distribution, page 10
τ	Denotes the period of the semi-infinite $\boldsymbol{H}^T$ matrix for LDPC-CCs, page 23
E_b	Energy-per-information bit, page 10
E_f	Signal energy for signal $f(t)$, page 9
I	Number of iterations in the decoders of LDPC-BCs and LDPC-CCs, page 21
J	Number of ones in a column of an $\boldsymbol{H}$ matrix in regular LDPC codes, page 15

K	Number of ones in a row of an $\boldsymbol{H}$ matrix in regular LDPC codes, page 15
M	Number of rows in an $\boldsymbol{H}$ matrix for a LDPC-BC, page 15
m_s	Memory order in LDPC-CCs, page 22
N	Typically used to denote a block size entering and leaving the FEC system, page 13
N_0	One-sided power spectral density of noise, page 10
P_f	Signal power for signal $f(t)$, page 9
R	Code rate of an FEC code, page 10
$r_u(t)$	Received information-bit symbol, page 30
$r_v(t)$	Received code-bit symbol, page 30
t	Time, page 22
V_{DD}	Positive power supply rail. 1.8-V for the 180 nm CMOS process, page 60
V_{SS}	Negative power supply rail and is typically grounded, page 60

List of Terms

a posteriori	Refers to a procedure or concept that is inductive or results from empirical evidence, page 18
bipartite graph	States that the nodes of the graph are of two distinctive types and the edges only connect the two different node types, page 16
channel capacity, C	Maximum rate of binary symbols (bits) that can be transmitted per second with a potential probability of error approaching zero. It is impossible to transmit at a rate higher than C without incurring errors, page 3
channel bandwidth	Indicates the range of frequencies that the channel can transmit with reasonable fidelity, page 2
codeword	Refers to the encoded block output by the FEC channel encoder, page 13
endec	A hardware entity comprising an **en**coder-**dec**oder combination. Similarly, the term "codec" is used to show the presence of a **co**der-**dec**oder combination in software entities, page 1
feed-forward system	Term given to a system that maintains a desired state based on changes to the environment. In contrast to a feedback system, a feed-forward system typically responds to a measured disturbance in a predefined way, i.e. how an FEC responds to varying SNRs yielding different BERs. Therefore, if and when errors are detected by the receiver, the redundant code is extracted from the message block and is used to predict and, hopefully, correct the discrepancy, page 13
full-duplex	Used to describe the communication process in systems. It means that data can be transmitted from point A to point B (and vice versa) simultaneously, page 3
Gaussian elimination	Common method used for solving matrix equations of the form $\boldsymbol{H} \cdot \boldsymbol{x} = \boldsymbol{y}$, page 18
GF(2)	Naming scheme used for finite fields in information and coding theory. The order, "(2)," represents the number of elements in a field. Operations in GF(2) are analogous to single-digit, binary arithmetic operations. For more information, see [7], page 4

Hamming distance	It is the distance (number of symbol errors) of the original transmitted codeword to the actual received codeword. This distance becomes inherently large as we attempt to detect and correct more errors but it is the minimum distance that we seek (lower bound). Good FEC codes ensure the Hamming distance between possible codewords have at least the Hamming distance arising from errors, page 22
ICFAALP2	CMC's code name, given to identify the chip in this thesis, page 42
IEEE 802.16	Formed in June 2001 and it refers to the certification mark for the IEEE 802.16 family of standards, now termed WiMAX. For more information, see `http://www.wimaxforum.org/`, page 5
inter-modulation noise	Occurs when two signals with certain frequencies are streamed through a non-linear device or component. The result will contain spurious frequency energy components that can be inside or outside the frequency band of interest [8], page 2
interleaver	Refers to a network of connections. In the case of LDPC codes, it refers to the connections from the variable nodes to the parity-check nodes and vice versa, page 36
netlist	Refers to a design that is realized through explicit instances of logic primitives and the interconnects between them. This is also called the structural or gate-level netlist, page 53
OSI model	This seven layer reference model is commonly used in communications and computer network protocol design. Not all protocols follow this hierarchy but it is a good reference that most adhere to. For more information, see [9], page 5
PHY	Refers to the physical layer in the standard, seven-layer OSI networking model, page 5
register-transfer level	A design that is implicitly modeled in terms of sequential (registers) and combinational logic to provide the desired data processing. Note that RTL descriptions can be converted into hardware via synthesis tools, page 56
repetition codes	One of the simplest FEC codes that works by transmitting each information bit i times so as to get a "majority vote" at the receiver. This restricts i to be an odd number to make the vote effective and renders the code rate to be $R = 1/i$, page 13

signal-to-noise ratio, SNR Power ratio of the transmitted signal to the channel noise such that $SNR = \frac{P_{signal}}{P_{noise}}$. Increasing signal power reduces the effects of channel noise so the information can be ascertained more reliably at the receiver. Also, a larger SNR allows transmission over a longer distance [10], page 2

syndrome A vector that identifies the location of the error in a message stream in systems employing FEC, page 14

thermal noise As per a Gaussian distribution, it is characterized by a uniform distribution of energy over the entire frequency spectrum. Prevalent in all communication systems and is caused by random electron motion. When molecules are heated above absolute zero, thermal noise will be present. Therefore, the more heat applied or generated, the greater the thermal noise level [8], page 2

translinear circuit A translinear device – for our purposes – is a transistor in which the drain current is an exponential function of the gate-to-source voltage. Most of the analog decoders today incorporate translinear devices. A Gilbert multiplier is a CMOS translinear circuit that upholds two principles: Sums of voltages are equivalent to the product of the currents; Operate in the subthreshold region to attain the exponential current to voltage relationship. For more information, see [4], page 36

Chapter 1

Introduction

The transfer of information – whether it be the transfer of voice information via cell phones, movie information on digital versatile discs (DVDs), etc. – is both ubiquitous and a necessity in today's society. This thesis proposes one of the first architectures for a forward error correction (FEC) channel encoding and decoding scheme employing low-density parity-check (LDPC) convolutional codes. Channel encoding and decoding is a way to send information from a source to a destination in a sufficiently error-free manner. Efficient implementations of these LDPC code endecs will allow us to incorporate them into next-generation communications standards. The purpose of this work is to show that a convolutional variant of these LDPC codes can also be realized in hardware and convey a different set of advantages for the same applications.

In Section 1.1, we describe the basics of a communication system and then lead into the motivation behind channel coding systems in Section 1.2. In Section 1.3, we present the outline for the remaining chapters in this thesis.

1.1 Digital Communication Systems

A fundamental model of a communication system, shown in Fig. 1.1, involves a source, a transmitter, a channel or medium, a receiver and a sink. Though the model is conducive to both analog and digital communication, we primarily deal with the digital domain in this thesis. The main objective of the model is to take input signals, distort or transform them, and then regenerate output signals that will hopefully be the same as your original input signals. For instance, Fig. 1.1 can be seen as two cell phone users communicating with each other:

1. Source: responsible for producing the information stream, i.e. source cell phone user talking.

2. Transmitter: modulates the information and prepares it for transmission, i.e. voice information is sent to a wireless network.

3. Channel: introduces random distortion to the transmitted signals. The channel is the main source of random noise and its consequences cannot be avoided, only mitigated.

4. Receiver: demodulates the information received and sends it to the sink, i.e. destination phone receives the information from the wireless network.

5. Sink: responsible for using the information collected from the receiver, to be used for further processing, i.e. cell phone user hears the message.

It should be noted that nearly all communication systems encompass these basic building blocks but the shear complexity of certain systems make it hard to determine an exact block distinction and separation.

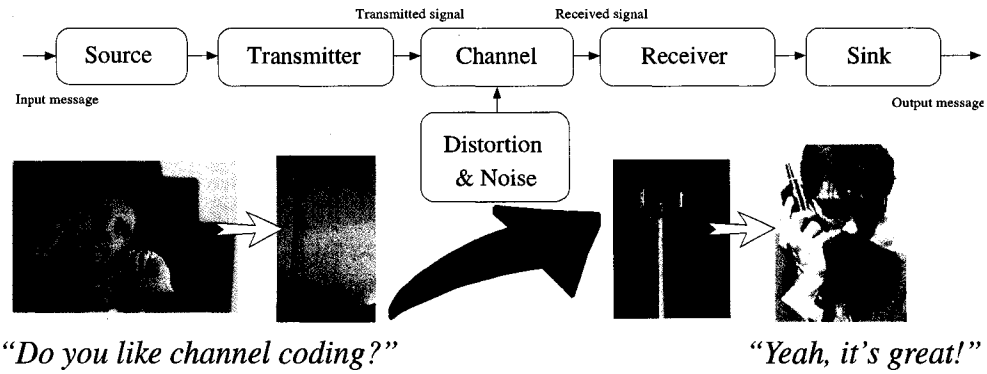


Figure 1.1: Basic digital communication system.

As the channel is inevitably unpredictable and potentially destructive (i.e. noisy), we require the receiver to do the best job it can to retrieve the transmitted signal(s). In other words, the amount of error correction necessary at the receiver is strongly dependent on the level of noise in the channel. This idea is conveniently quantified by the signal-to-noise ratio (SNR), which is the power ratio of the transmitted signal to the background noise in the channel. There are some very simple ways to improve the SNR and thus, alleviate erroneous transmissions. Although these methods are effective and straightforward, they do have several disadvantages that make them infeasible [11]:

1. Increase signal power: the most obvious way is to increase the transmitter power when assuming the channel characteristics immutable. However, this is not always possible in practice since most communication devices are power-limited and have no way for increasing the power. In addition, channel bandwidth and signal power are exchangeable. That is, an increase in signal power buys a reduction in channel bandwidth and the converse is also true. However, the former sees a smaller benefit than the latter (or converse) and hence in practice, channel bandwidth is usually traded to reduce signal power [10].

2. Decrease transmission noise: if the communication device is power-limited, then the next logical choice would be to minimize the noise itself. This can be accomplished by turning off the sources that can potentially cause interference. However, this is a daunting task and is literally implausible. The only eliminations or minimizations that can be made are for thermal and inter-modulation noise, which are under the designer's control. In short, we must still face the consequences of a volatile environment and research other ways to improve the SNR.

3. Diversify environments: if the signal integrity is fading or deteriorating, system redundancy can be used to effectively boost the perceived SNR. One example could be to use two receivers from two different locations to receive the same information, thus, seeing a higher SNR. The two receivers then work together to decipher the transmitted information with a reduced error probability. The main objective of diversity is to increase SNR by utilizing redundancy, and in this case, at the expense of higher resource requirements. Alternative forms of this technique are still widely used.

4. Retransmit information: considered to be another form of diversity and it requires bidirectional communication. Each transmitted block is checked by the receiver for errors. In case of errors, a request is made for retransmission while receiving other blocks concurrently. It is evident that if the SNR is high, there is little retransmission and throughput is high; however, if the SNR is low, the retransmission overhead becomes significant and renders the system inefficient. This technique is wasteful on several accounts but the key idea is to make use of redundancy to combat noise.

The above methods are selfish when applied alone but when put together, they still prove to be quite resourceful. One such example is in [12], where an algorithm is devised to reduce signal power in short-range wireless networks but make use of block retransmission. As we advance technologically, we uncover more methods to drive the probability of error lower.

Claude Shannon, known to be the father of practical information theory, revolutionized the telecommunications industry when he conceptualized "error-free" digital communication over arbitrarily noisy channels in his renowned 1948 article [13]. In summary, Shannon proves the maximum possible efficiency of error-correcting coding techniques at the receiver versus the noise inherent in the channel. He states that if there exists a noisy channel with a channel capacity C and information that is transmitted at a rate R such that $R < C$, then there exists an error-correcting code (ECC) to make the probability of error for the received information arbitrarily small. Conversely, if $R > C$, information that is sent through the channel cannot be guaranteed to be received reliably [10, 14, 15]. Shannon does not mention

the rare case when $R = C$ nor does he offer insight on how to construct these ECCs. With this notion, the quest for attaining suitable ECCs to fix errors caused by channel imperfections began. This came to be known simply as channel coding or error-control coding.

1.2 Channel Coding

The basic communication system previously shown in Fig. 1.1 can now be modularized even further to include source and channel coding, which is illustrated in Fig. 1.2. Without going into much detail, source encoding and decoding can be either lossy or lossless. An real-life example of a lossless scheme is the Lempel-Ziv-Welch (LZW) algorithm used in today's portable document format (PDF) documents [16]. Similarly, one lossy scheme that is prevalent among users is the joint photographic experts group (JPEG) picture standard [17]. Although source encoding (and decoding) is not the subject of this thesis, it is worth mentioning that it deals with the compression and decompression of the source data itself and has nothing to do with the channel.

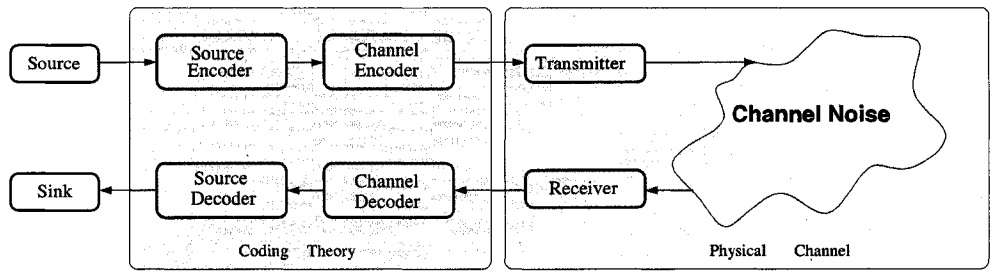


Figure 1.2: Basic model for modern-day digital communications systems.

Channel coding makes use of redundancy and is illustrated in Fig. 1.3. This concept involves a simple example of a single parity-check code, in which an additional bit is appended to an information block to ensure the total number of binary 1's are even. If a single error occurs at the receiver (i.e. a violation of the parity-check constraint), then a retransmission of that block is requested. The natural extension of this concept leads to the construction of complex coding procedures that can detect and correct several bits [10].

There are two major classes of error control codes: linear and non-linear. However, linear codes are studied the most due to favorable properties such as simpler encoding and decoding methods and straightforward algebraic descriptions [7]. Here, "linear" means creating redundancy bits by combining data bits linearly and we limit our field of view to binary codes over GF(2) (Galois field). Though there are several code types, we limit our analysis only to linear block and convolutional types for LDPC coding, which will be discussed in detail in Chapter 2.

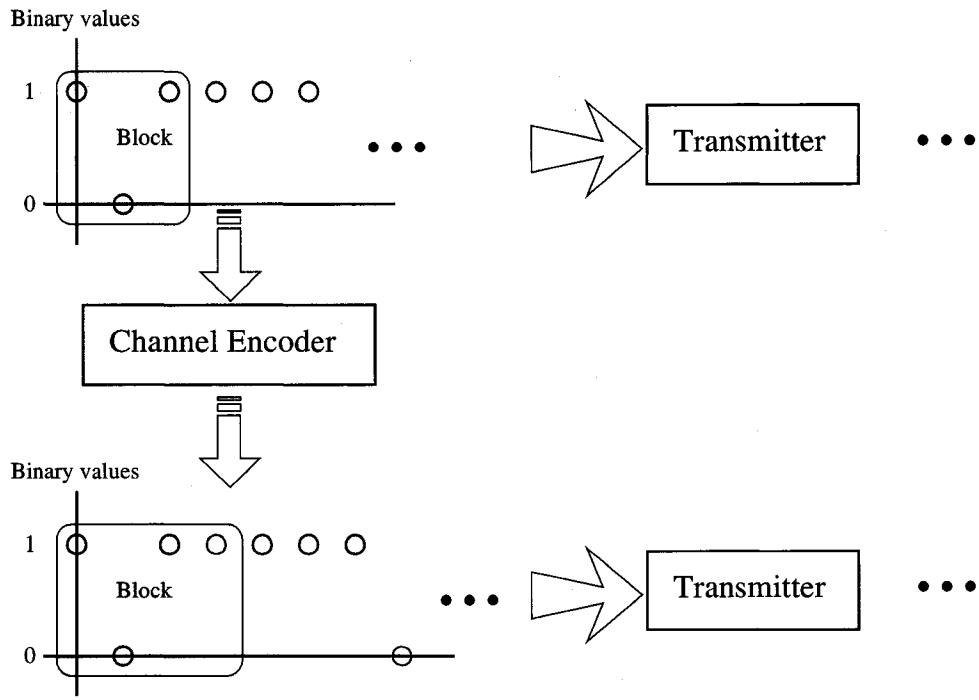


Figure 1.3: Visual example of a simple ECC with an odd parity-bit append.

A communication system with channel coding, like in Fig. 1.2, can be easily described in theory but its real-life application can sometimes be hard to objectify. Applications of linear block codes include deep-space and mobile communications and digital audio recording – compact disc (CD) technology [18], for example. However, to demonstrate how LDPC block codes are used in a more recent example, we present a brief case study of the worldwide interoperability for microwave access (WiMAX) certification, part of the institute of electrical and electronics engineers (IEEE) 802.16 standard.

1.2.1 Brief Case Study: IEEE 802.16 Standard

This upcoming broadband, air interface standard is expected to overtake the IEEE 802.11 standard because of its extended range and significantly higher data rates. Of importance here, is the IEEE 802.16e amendment that concentrates on the physical (PHY) and medium access control (MAC) layers for fixed and mobile wireless communications [19]. In other words, WiMAX is a connection-oriented technology affecting only the data link layer and the PHY layer of the open systems interconnection (OSI) model – commonly known as the OSI seven layer model as shown in Fig. 1.4 [9].

The MAC is a sublayer in the data link layer directly above the PHY layer, which is the actual physical medium for sending and receiving data. The MAC sub-

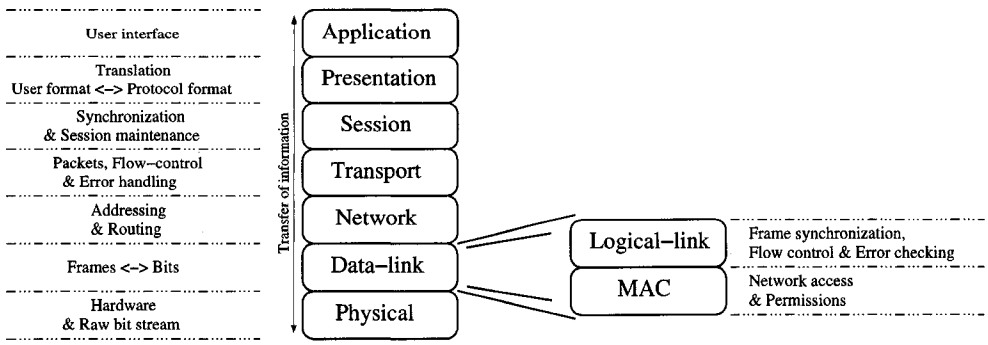


Figure 1.4: Standard OSI reference model for communications and computer network protocol design.

layer must provide access to a particular physical encoding and transport scheme. However, it is in the PHY layer where error correction is implemented. Fig. 1.5 shows a typical WiMAX base station as a transmitter and receiver.

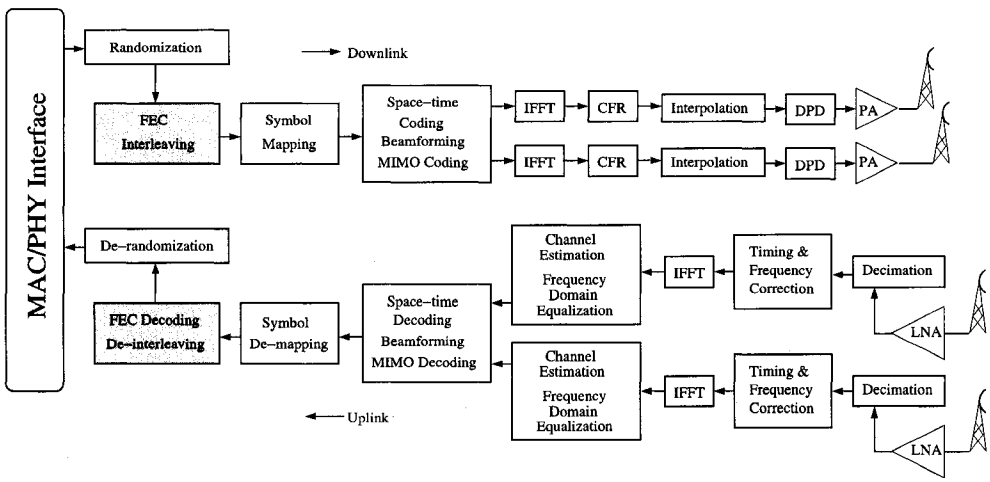


Figure 1.5: High-level block diagram of the PHY layer in a typical WiMAX base station [2].

As per the IEEE 802.16e amendment and Fig. 1.5, several channel coding techniques have been considered for the FEC block, all of which must follow four distinct steps [19, 20]:

1. Randomization (de-randomization): a pseudorandom noise (PN) sequence generator is used to randomize (de-randomize) each data block that is to be channel coded. This effectively scrambles (de-scrambles) the signal.

2. FEC (endec): these include tail-biting or zero-tailed convolutional coding, convolutional Turbo and LDPC coding. Currently, LDPC coding is optional but it is a choice that is not overlooked.

3. Interleaving (de-interleaving): channel-coded data is then interleaved (de-interleaved) by first applying a "scattering" method and then performing a least-significant-bit (LSB)/most-significant-bit (MSB) switch.

4. Modulation (de-modulation): orthogonal frequency division multiple access (OFDMA) is a system multiplexing standard, where quadrature phase shift keying (QPSK) and 16/64 quadrature amplitude modulation (QAM) are available.

The IEEE 802.16e standardization group promises to support mobile communications at speeds of approximately 40 to 50 mph for data rates of 5 Mbps or better. LDPC encoding and decoding are considered to provide "error-free," point-to-point communication, not only in WiMAX but in several other standards as well.

1.3 Outline of the Thesis

The main subject of this thesis is to showcase a recently discovered class of LDPC codes, known as LDPC convolutional codes, and its implementation specifics. In doing so, Chapter 2 elaborates on the theory of channel coding, revolving around LDPC codes and its attributes. Chapter 3 is solely dedicated to the analysis of the LDPC convolutional variant. Chapter 4 discusses prior work in the area of FEC and LDPC codes. This leads into the application specific integrated circuit (ASIC) implementation of the first known LDPC convolutional code encoder and decoder. We provide an in-depth analysis of the chip's architecture and report expected and measured results in Chapters 5 and 6, respectively. We finally conclude and state the directions of future research for this area in Chapter 7.

Chapter 2

Background to Channel Coding

It should be clear by now that channel coding is an essential part of today's communication systems. Though it is not what is showcased in a complex system, without it, reliable information could not propagate to the other processing units. In this chapter, we will take a closer look into channel encoding and decoding, with primary focus placed on LDPC codes. In Section 2.1, we present the theory required to understand most of what happens between the channel encoder and decoder. Then we introduce FEC that lead into concepts in iterative decoding theory utilized in LDPC block coding in Sections 2.2 and 2.3.

2.1 Terms and Concepts

Throughout this thesis, basic concepts are used to describe values and/or components in a system. Though the application may be different, the definition remains the same. Therefore, brief definitions and descriptions are provided for those underlying concepts.

2.1.1 Signal Energy & Power

A measure of signal energy and power are mandatory in all systems whether it be the measure of the wanted or unwanted parts of the signal. Conventionally, signal energy in the time domain, E_f, is defined as

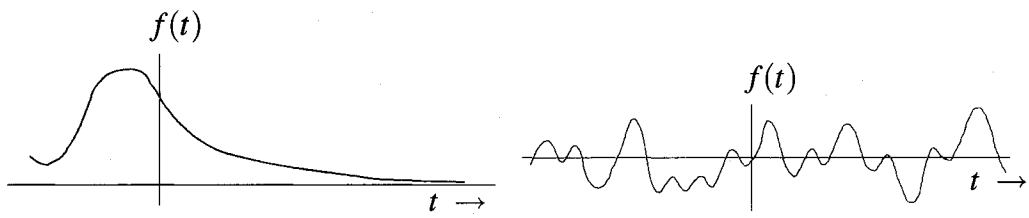
$$E_f = \int_{-\infty}^{\infty} |f(t)|^2 dt \tag{2.1}$$

and similarly, signal power, P_f, is defined by

$$P_f = \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{T/2} |f(t)|^2 dt. \tag{2.2}$$

Signal power, if it exists, is a more meaningful measure as it is a time average of signal energy and hence, the entity is either periodic or has statistical regularity [21]. The units of signal energy and power depend on the input signal, $f(t)$:

1. If $f(t)$ is a voltage signal, E_f has units $V^2 \cdot s$ (Volts-squared-seconds) and P_f has units V^2 (Volts-squared).
2. If $f(t)$ is a current signal, E_f has units $A^2 \cdot s$ (Amperes-squared-seconds) and P_f has units A^2 (Amperes-squared).



(a) Energy signal (b) Power signal

Figure 2.1: Example signal representations.

Figs. 2.1(a) and 2.1(b) illustrate an example of an energy and power signal, respectively. An important attribute to observe is that since power averages over an infinitely large interval, a signal with finite energy has zero power and a signal with finite power has infinite energy. They are mutually exclusive but there exist signals that are neither power nor energy signals [21].

2.1.2 Signal-to-Noise Ratio (SNR)

SNR is a ratio of the signal power to noise power. It is a purposive measure that is widely used in communication theory to quantify noise from a deterministic viewpoint. It is typically measured at the receiving end of the communication system before signal detection and it allows us to evaluate and anticipate the perilous effects of noise [22]. SNR, in decibels (dB), is defined as

$$SNR = 10 \cdot \log_{10} \left(\frac{signal\ power}{noise\ power} \right) dB. \tag{2.3}$$

SNR can also be expressed as a ratio of signal voltage to noise voltage if we assume the inputs signals are measured across the same resistance ($R_1 = R_2$). Thus, in dB,

this is shown by

$$\begin{aligned} SNR &= 10 \cdot \log_{10}\left(\frac{signal\ power}{noise\ power}\right) \\ &= 10 \cdot \log_{10}\left(\frac{(V_{signal})^2/R_1}{(V_{noise})^2/R_2}\right) \\ &= 10 \cdot \log_{10}\left(\frac{V_{signal}}{V_{noise}}\right)^2 \\ &= 20 \cdot \log_{10}\left(\frac{signal\ voltage}{noise\ voltage}\right) dB. \end{aligned} \tag{2.4}$$

To put this in perspective, the error probability or SNR of a coded communication system is commonly expressed as a ratio of energy-per-information bit (E_b) to the one-sided power spectral density (N_0) of channel noise [23]. The E_b is obtained by dividing signal power, which is conventionally denoted σ_s^2, by the code rate, R. N_0 is defined by the channel model used in the communication system; see Section 2.1.3 for more information. The code rate, R, is defined as per Section 2.1.6.

$$SNR = \frac{E_b}{N_0} \tag{2.5}$$

$$E_b = \frac{\sigma_s^2}{R} \tag{2.6}$$

N_0 = dependent on the channel model

2.1.3 Channel Model

There exist several techniques and methods for modeling the channel or transmission medium. Of them all, the most prominent model used in academia, and in this thesis, is the additive white Gaussian noise (AWGN) channel. Fig. 2.2 shows the basic structure of an AWGN channel. The noise incurred is specified by $N(\mu, \sigma_n^2)$, meaning the distribution of the noise has a mean of μ and a noise variance of σ_n^2. Note that in a *normal* Gaussian distribution, we have $N(\mu = 0, \sigma_n^2 = 1)$.

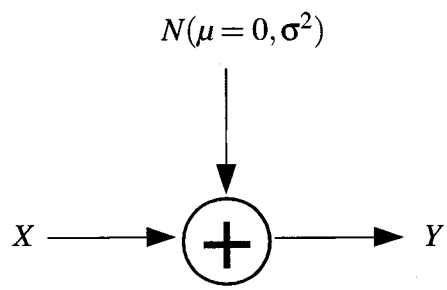


Figure 2.2: AWGN channel model.

For this channel type, the noise variance is related to noise power spectral density by

$$\sigma_n^2 = \frac{N_0}{2}. \tag{2.7}$$

Therefore, modeling different SNRs can be accomplished by varying the σ_n (standard deviation) in the noise probability distribution.

2.1.4 Modulation

The modulation scheme is typically considered to be part of the channel and is responsible for physically transmitting and receiving signals by varying a periodic waveform. Demodulation is the inverse operation of modulation but at the receiver. The aim of digital modulation is to convey the digital bitstream. Phase shift keying (PSK) is just one form of angle-modulated, constant-amplitude digital modulation that takes a binary digital signal as input. Binary PSK (BPSK) is an M-ary system where $M = 2$ and two output phases are possible for a single carrier frequency: one of the phases represents a logic '0' and the other a logic '1'. Fig. 2.3 visually shows an ideal BPSK truth table [24].

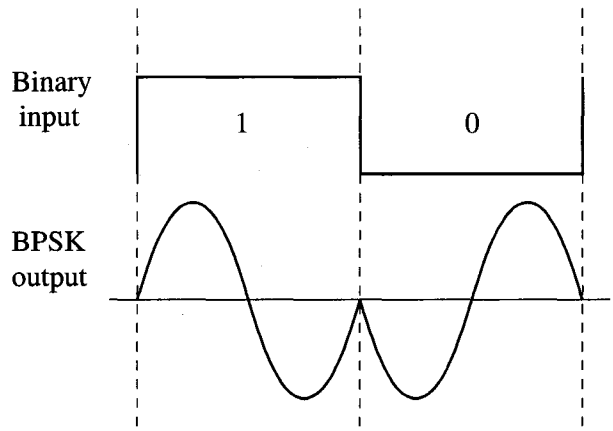


Figure 2.3: Ideal output phase-versus-time relationship in BPSK modulation.

In this thesis, we focus on BPSK modulation as it fits well with the FEC decoding scheme of interest.

2.1.5 Performance Metric

The performance of a coded communication system is conventionally measured by its probability of decoding error for a perceived SNR over its uncoded counterpart that transmits information at the same rate – also known as "coding gain." This decoding error can be measured in two ways: bit error rate (BER) probability or frame error rate (FER) probability. FER is defined as the probability that a decoded

frame is in error at the output of the decoder. Similarly, BER is the probability of an information bit in error at the output of the decoder. For example, a BER of 10^{-5} means that one in every 100,000 decoded information bits is in error. Coding gain is the other metric defined as the reduction in SNR required to achieve a certain BER/FER for a coded versus uncoded communication system. This SNR reduction is not indefinite and thus, the ceiling on coding gain is defined by the capacity of the channel.

In any case, the communication system should be designed to keep both of these error probabilities as low as possible [23]. These performance metrics are exemplified in Fig. 2.4.

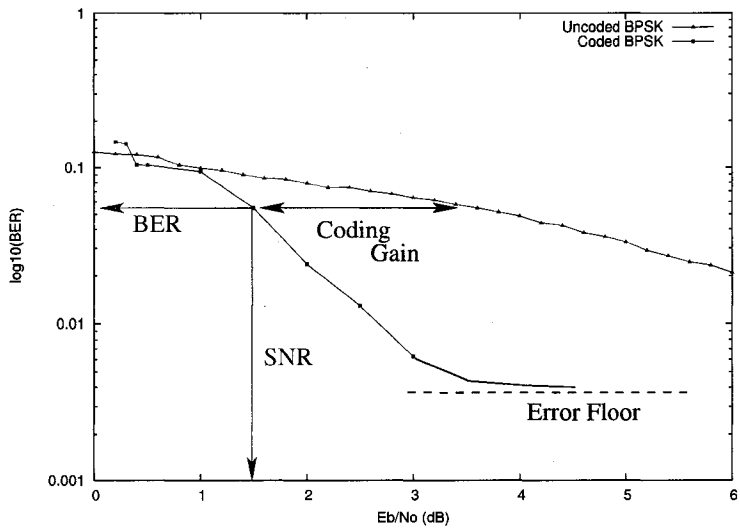


Figure 2.4: Example BER vs. SNR graph showing an uncoded and coded plot using BPSK modulation.

2.1.5.1 Error Floor

Also illustrated in Fig. 2.4, an error floor is a region in the performance graph where the slope of the error probability begins to flatten. This can mean the area where the slope at higher SNRs has decreased relative to the slope at lower SNRs. Error floors are typically unwanted as they decrease the strength or usefulness of the FEC code [23].

2.1.6 Code Rate

The rate of the FEC code is defined as R and it quantifies what amount of the total information sent through the channel is useful information, i.e. non-redundant. Therefore, if the code rate is k/N, this means that for every k information bits (message), there are N bits produced by the channel encoder and hence, $N-k$

bits represent the redundant data. Higher code rates mean fewer code-bits present to combat noise. Whereas, lower code rates mean more code-bits present in the stream and hence, more data protection. Moreover, code rate, R, is typically a number between 0 and 1.

2.2 Forward Error Correction

FEC attempts to correct errors using one-way communication, i.e. there is no feedback from the decoder and thus, is also referred to as a "feed-forward system." FEC is used in systems where it is impractical to request a retransmission of the corrupted message block. Therefore, if and when errors are detected by the receiver, the redundant code is extracted from the message block and is used to predict and hopefully correct the discrepancy.

As mentioned earlier, we only concentrate on linear FEC codes, with primary focus on LDPC block codes (LDPC-BCs) and convolutional codes (LDPC-CCs). Since the information in most digital communication systems are coded in binary (digital), we also limit ourselves to operations in GF(2). Before LDPC codes are discussed, it is worthwhile to note the first effective set of linear block codes after Shannon's seminal article in 1948 [13], i.e. Hamming codes [25].

2.2.1 Hamming Codes

In 1950, Richard W. Hamming introduced an encoding and decoding scheme that was able to detect and correct single-bit errors in message blocks. His work marked the beginning of the search for FEC codes that can approach the Shannon limit for channel capacity. Hamming codes can be used to visually demonstrate the use of parity bits in a more sophisticated manner to improve the code rate over existing methods, such as with repetition codes.

For example, a $(N = 7, k = 4)$ Hamming code with an information block of size k,

$$\boldsymbol{u} = (u_0, u_1, u_2, u_3) \tag{2.8}$$

is mapped by the channel encoder into another block of size N,

$$\boldsymbol{v} = (v_0, v_1, v_2, v_3, v_4, v_5, v_6), \tag{2.9}$$

where $\boldsymbol{v}$ is the block that is transmitted over the channel and is referred to as the

codeword [26]. This resulting codeword has the relationship to its input given by

$$\begin{aligned} v_0 &= u_0, \\ v_1 &= u_1, \\ v_2 &= u_2, \\ v_3 &= u_3 \\ v_4 &= u_0 \oplus u_2 \oplus u_3, \\ v_5 &= u_0 \oplus u_1 \oplus u_3, \\ v_6 &= u_0 \oplus u_1 \oplus u_2, \end{aligned} \tag{2.10}$$

where "$\oplus$" denotes modulo-2 addition. This relationship is more appropriately described in a row-wise, $k \times N$ matrix, called the *generator matrix*, $\boldsymbol{G}$,

$$\boldsymbol{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{2.11}$$

and $\boldsymbol{v}$ can be obtained by the vector multiplication

$$\boldsymbol{v} = \boldsymbol{u} \cdot \boldsymbol{G}. \tag{2.12}$$

Similarly at the receiver side, the source message can be retrieved if

$$\boldsymbol{r} \cdot \boldsymbol{H}^T = \hat{\boldsymbol{s}}, \tag{2.13}$$

where $\boldsymbol{r}$ is the received message including channel noise, $\hat{\boldsymbol{s}}$ is the *syndrome* vector and $\boldsymbol{H}$ is the *parity-check matrix* satisfying the constraint

$$\boldsymbol{G} \cdot \boldsymbol{H}^T = 0. \tag{2.14}$$

Therefore, if $\boldsymbol{r}$ contains at most one error, then the syndrome $\hat{\boldsymbol{s}}$ will be a non-zero vector specifying the location of the bit in error. However, if $\boldsymbol{r}$ contains no errors, then $\hat{\boldsymbol{s}}$ will be $\mathbf{0}$, the zero vector.

For example, if $\boldsymbol{u} = (1,0,1,1)$ is transmitted using $\boldsymbol{G}$ in (2.11) and an $\boldsymbol{H}^T$ matrix is given by

$$\boldsymbol{H}^T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.15}$$

then $\boldsymbol{v} = (1,0,1,1,1,0,0)$. Now suppose we have a corrupted bit in the received stream, then $\hat{\boldsymbol{s}}$ holds a unique value to identify that error. Table 2.1 shows all the

$\hat{s}$	Location of Bit-error
[0,0,0]	No Error
[1,1,1]	v_1
[0,1,1]	v_2
[1,0,1]	v_3
[1,1,0]	v_4
[1,0,0]	v_5
[0,1,0]	v_6
[0,0,1]	v_7

Table 2.1: Shows all possible $\hat{s}$ values and error positions for a (7,4) Hamming code.

unique values for this (7,4) Hamming code. Then to correct the single-bit error, we simply flip that bit.

If multiple errors exist in r, it is also possible to satisfy (2.13) yielding $\hat{s} = 0$. When this occurs, the errors are referred to as *undetectable errors*. Therefore, to detect and correct multiple bit errors, more complex FEC techniques are required.

2.3 LDPC Codes

Since Hamming codes, the increasing need for reliable transmission led to the discovery of LDPC codes in the early 1960s by Robert Gallager [27]. However, these codes were not pursued further at that time because of the prohibitively large computational demands of the decoding algorithm. The term "low-density" is characteristic of its parity-check matrix in that it contains fewer non-zero elements in comparison to its zero elements. The renewed interest in LDPC codes over the past decade occurred when it was realized that they were, in fact, among the most powerful class of ECCs known to date.

2.3.1 LDPC-BCs

There exist LDPC-BCs today that can reach within 0.0045 dB of the capacity of a binary, antipodal-modulated AGWN channel [28]. Their encoding and decoding schemes are relatively complex and can be described via a parity-check matrix (like all other linear codes) or be represented graphically.

2.3.1.1 Matrix & Graphical Representation

As an example, a $(N = 16, J = 3, K = 6)$ LDPC-BC is represented by the $\boldsymbol{H}$ matrix in (2.16).

$$\boldsymbol{H} = \begin{bmatrix} 1\,1\,0\,1\,0\,0\,0\,0\,1\,0\,0\,1\,1\,0\,0\,0 \\ 0\,1\,1\,0\,1\,0\,0\,0\,1\,1\,0\,0\,0\,0\,1\,0 \\ 0\,0\,1\,1\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,1 \\ 0\,0\,0\,1\,1\,0\,1\,0\,0\,1\,1\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0\,1\,0\,0\,0\,0\,1\,1\,1\,0 \\ 1\,0\,0\,0\,0\,1\,1\,0\,1\,0\,1\,0\,0\,0\,0\,1 \\ 0\,1\,0\,0\,0\,0\,1\,1\,0\,0\,0\,1\,0\,0\,1\,1 \\ 1\,0\,1\,0\,0\,0\,0\,1\,0\,0\,1\,1\,0\,1\,0\,0 \end{bmatrix} \tag{2.16}$$

With respect to (2.16), we can now define J and K as the number of ones in each column and row, respectively. We also denote M to represent the number of rows and N to be the number of columns (total block length) in the $\boldsymbol{H}$ matrix. Thus, the code rate is given by

$$R = \frac{N-M}{N}. \tag{2.17}$$

In 1981, Michael Tanner gave LDPC codes a graphical point of view that had the ability to provide a full representation of the code [29]. Fig. 2.5 shows the Tanner graph representation of the $\boldsymbol{H}$ matrix in (2.16).

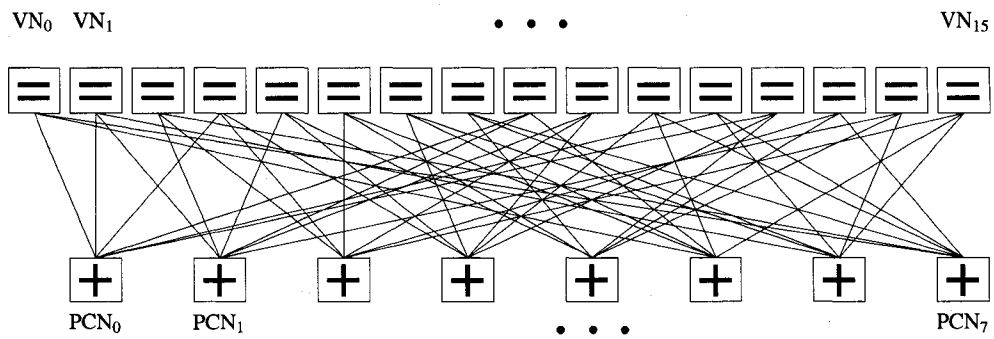


Figure 2.5: Graphical representation of the $\boldsymbol{H}$ matrix, using parity-check nodes ("+") and variable nodes ("=").

Tanner graphs are also called *bipartite graphs*, which means that the graph is split into two distinctive types and the edges only connect the two different types [30]. Thus, LDPC-BC bipartite graphs comprise two node types: variable nodes (VNs) and parity-check nodes (PCNs), associated with "=" and "+," respectively.

As illustrated in Fig. 2.5, the creation of this bipartite graph is straightforward. It consists of M PCNs that correspond to M rows in the $\boldsymbol{H}$ matrix and similarly, N VNs that correspond to the N columns of the same matrix. Therefore, the parity-check node, PCN_m, is connected to variable node, VN_n, if and only if there is a '1' in element $\boldsymbol{H}(m,n)$ for $m = 0 \cdots M-1$ and $n = 0 \cdots N-1$.

2.3.1.2 General Attributes

There are a few common terms used to describe the LDPC code's structure and behavior.

Systematic versus non-systematic codes: in systematic codes, the input data physically appears in the encoded data. In a non-systematic code, the encoded data does not contain the input data. Systematic codes are ideal when the probability of error is relatively low. In such cases, the original data can be recovered from the received data knowing well that the coded parts are just for data correctness. Non-systematic codes can be fairly flexible in that they can exploit the use of all redundant (coded) data. This typically requires the decoder to be more computationally complex and thus, seem appropriate for channels with much higher probability of error – wireless channels, for example. In this thesis, only systematic LDPC codes are considered because of their relative ease in implementation.

Regular versus irregular codes: an LDPC code is considered "regular" if the weights J and K are constant for each column and row (respectively) in the parity-check matrix $\boldsymbol{H}$. Conversely, if either of the weights J or K vary from column-to-column or row-to-row (respectively), then the code is called "irregular." It is also possible to see this behavior in the bipartite graph - if the number of incoming edges across all variable nodes are the same and similarly, for all parity-check nodes, then the code is regular. Moreover, the code rate R may alternatively be calculated using these weights and is defined by

$$R \geq 1 - \frac{J}{K}. \tag{2.18}$$

It is known that some of the most powerful, capacity-approaching LDPC codes are currently irregular codes [28, 31]. Though powerful, irregular codes are extremely challenging to build and realize in hardware. In this thesis, we deal only with regular LDPC codes.

Soft-decision versus hard-decision decoding: soft-decision decoders accept "soft-input" from the channel and produce "hard-output," i.e. binary zeroes or ones. However, hard-decision decoders accept "hard-input" and yield "hard-output." Soft-input can be probabilities or estimates of the received symbols and it can be used to produce hard-output estimates of the correct symbols [7]. It is known that soft-decision decoders offer better performance over hard-decision decoders but with the disadvantage of higher decoder complexity [23]. In this thesis, we consider only soft-decision decoding algorithms and architectures.

2.3.1.3 Code Construction & Encoding

Although Gallager did not provide a specific method for constructing "good" LDPC codes, he did propose a general method for constructing a class of pseudorandom LDPC codes. It has been found that good, large block-length LDPC codes are namely computer generated or random in nature [23]. Hence, their encoding is very complex due to the lack of structure or regularity. However, several effective methods have been noted for constructing LDPC-BCs that still approach channel capacity. For more information on construction techniques, see [28, 31, 32].

The encoding operation is similar to that of Hamming codes mentioned in Section 2.2.1. From a constructed $\boldsymbol{H}$ matrix, we derive the generator matrix $\boldsymbol{G}$ via Gaussian elimination. It turns out that $\boldsymbol{G}$ is not "low-density" relative to its $\boldsymbol{H}$ matrix counterpart and hence, encoding is also relatively complex [23, 26, 30]. Early work showed that encoding could only be done in quadratic time, $\mathcal{O}(N^2)$, with respect to the block length N. However, recent work has shown that encoding (even for large codes) can be completed in linear time, $\mathcal{O}(N)$ [33].

2.3.1.4 Decoding

The decoding operation is where most of the complexity arises and there is still an ongoing search to ease the requirements for this task. There are several algorithms for the decoder, namely: majority-logic (MLG) decoding, bit-flipping (BF) decoding, weighted BF decoding, *a posteriori* probability (APP) decoding and an iterative-based decoding based on belief propagation (BP), also known as the sum-product algorithm (SPA). Of the five methods, SPA is the soft-decision algorithm known to yield good error performance and still be computationally tractable [23].

SPA is also, interchangeably, referred to as the message-passing algorithm (MPA) because the algorithm passes soft-information back and forth between the node types in the bipartite graph representative of the $\boldsymbol{H}$ matrix. It is important to note that the SPA is one of several possible MPAs. This iterative decoding is extremely efficient with LDPC codes as it processes the received symbols iteratively to improve the reliability of each decoded symbol. The reliability or probability of a symbol can be expressed as a log-likelihood ratio (LLR), denoted by ℓ, which is the natural logarithm of the ratio of the probability that the corresponding bit is a '0' to the probability that the bit is a '1.' Working in the logarithmic domain converts multiplications into additions and divisions into subtractions, hence simplifying calculations and reducing complexity. With regard to the bipartite graph and the parity-check matrix $\boldsymbol{H}$ seen in Fig. 2.5 and (2.16), respectively, the MPA algorithm is as follows [1, 27]:

1. Initialize all VNs and their outgoing messages (edges on the bipartite graph) to the values of their incoming channel LLRs. From Fig. 2.6, $\ell_n^{(0)}$ for $n = 0 \cdots N-1$ marks the LLR received from the channel and is a

value that does not change for the entire decoding process for that block.

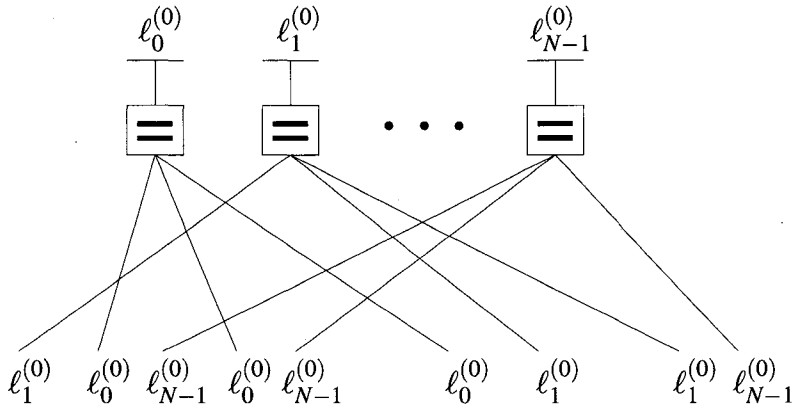


Figure 2.6: Initialization phase of the MPA for LDPC-BCs.

2. Send the messages from all VNs to the PCNs along the connected edges of the bipartite graph.

3. Perform a soft-input parity-check operation for the incoming LLRs along the edges. Fig. 2.7 depicts the computation for PCN, PCN_m, as $m = 0 \cdots M-1$ and $k, n = 0 \cdots N-1$. This composite operation is split into a sign and magnitude component:

   - *Sign operation:* perform an XOR operation on the sign bits of all the incoming LLRs to form the parity-check result for the m^{th} row in the $\boldsymbol{H}$ matrix. Then form the sign bit for each outgoing LLR message for each edge on the graph by XORing the sign of the incoming VN LLR message corresponding to that edge and m^{th} row's parity-check result.
   - *Magnitude operation:* the magnitude portion or the reliability measure of an outgoing LLR message is calculated using

$$\ell_{mk}^{*} = 2\tanh^{-1}\left(\prod_{\forall n, \boldsymbol{H}(m,n)=1, n\neq k} \tanh\left(\frac{\ell_{mn}}{2}\right)\right), \qquad (2.19)$$

     where ℓ_{mk}^{*} is the magnitude portion of the outgoing LLR message from PCN_m to VN_k and ℓ_{mn} is the magnitude of the incoming LLR message sent to PCN_m from VN_n.

4. Pass the LLR messages from the PCNs back to the VNs along the edges of the bipartite graph to begin the next iteration cycle.

5. At VN_n, for $n = 0 \cdots N-1$, all the incoming LLR estimates from the PCNs are then summed along with the original channel LLR, $\ell_n^{(0)}$. In

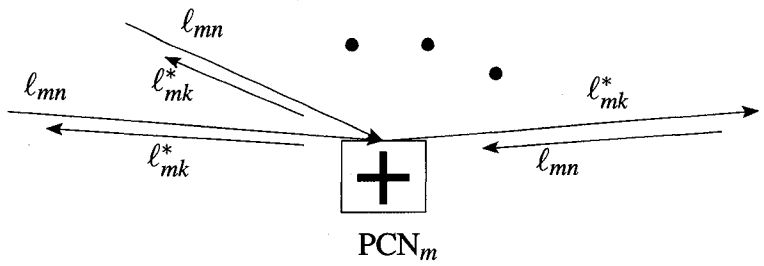


Figure 2.7: PCN computation in the MPA.

BPSK, the decoded bit would be the sign bit of the previous summation. However, if the outgoing LLR message for the next decoding iteration is to be formed using the same summation, then this can be realized by subtracting the contribution from each edge from the total sum. This relationship is equivalent to

$$\ell^*_{nk} = \ell^{(0)}_n + \sum_{\forall m, \boldsymbol{H}(m,n)=1, m\neq k} \ell_{nm}, \tag{2.20}$$

where ℓ^*_{nk} is the outgoing LLR message from VN$_n$ to PCN$_k$ and ℓ_{nm} is the incoming LLR message sent from PCN$_m$ to VN$_n$ for $n = 0 \cdots N-1$ and $m,k = 0 \cdots M-1$. As pointed out by Fig. 2.8, this is similar to the parity-check operation in that the edge updated does not include the incoming LLR message on that corresponding graph edge. $\ell^{(0)}_n$ is never removed from the group sum.

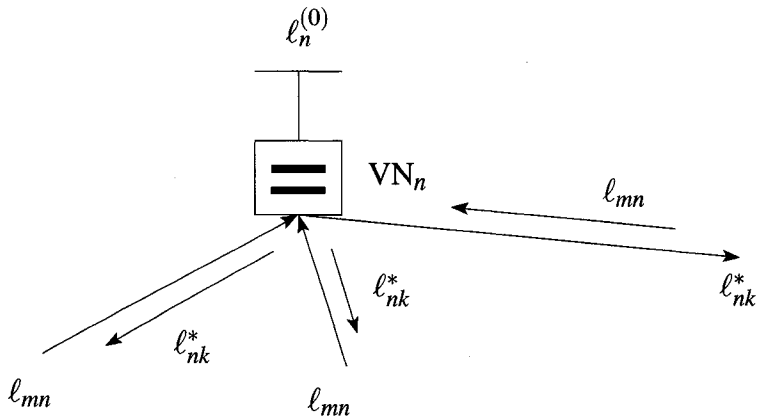


Figure 2.8: VN computation in the MPA.

6. Steps 2-5 are repeated until a termination condition is satisfied. There are several possibilities for terminating conditions:

   - The estimated decoded block, $\hat{\boldsymbol{u}}$, satisfies the condition

   $$\hat{\boldsymbol{u}} \cdot \boldsymbol{H} = \boldsymbol{0}. \tag{2.21}$$

This could mean an infinite number of iterations if the codeword could not converge to a solution.

- The intermediate messages in the decoder satisfy all of the parity-checks. This exits the decoder with the first codeword that meets all the constraints, but that may not necessarily be the correct codeword.
- Stop the decoding process after I number of iterations. This has the potential for degrading error performance and/or resulting in an incomplete solution but it is the most pragmatic approach.

For a much more elaborate consultation of the MPA or SPA, see [23, 27, 34].

2.4 Summary

FEC is a must in today's modern world since short- and long-distance communication has become an involuntary standard. Markets such as broadband wireless and mobile networks operate in noisy environments and need powerful error correction in order to improve reliability and data rates. LDPC-BCs are known to be among the most powerful, capacity-approaching class of FEC today, posing strong competition against Turbo codes. Due to their capabilities, LDPC-BCs are becoming the FEC code of choice in next-generation communications standards such as the IEEE 802.16e (WiMAX) [19], 802.11n (Wi-LAN) [35] and 802.3an (10GBASE-T) [36]. The major benefits from using LDPC-BCs is a performance gain that can result in lower transmit power, higher data throughput, longer transmit distance, and increased reliability of the communication link. When transmit power is limited, the extra coding gain of LDPC codes can make the difference between reliable communication and no communication. In Chapter 3, we introduce the convolutional alternative, also in the LDPC code family, that offers a different outlook for the same FEC applications.

Chapter 3

LDPC Convolutional Codes

Block codes encode and decode using only the data from a fixed-size block. The concept of convolutional coding was first introduced by Peter Elias in 1955 and was devised to be an alternative to block codes [37]. The basic idea of convolutional codes is that outputs of the encoder for a given time unit depend not only on the inputs at the that time unit but also on previous inputs. This introduced the concept of *memory order*, m_s, where inputs would remain in the encoder for an additional m_s time units after entering. Dissimilar to block codes, the large minimum Hamming distances and low error probabilities are plausible not by increasing the input message size and block size, but by increasing the memory order, m_s [23].

In Section 3.1, we introduce LDPC-CCs and their distinct qualities. This is followed by Sections 3.3 and 3.4, where their encoding and decoding operations are explained, respectively. Lastly, we compare their coding performance with respect to LDPC-BCs in Section 3.4.1.

3.1 Overview

Another subclass of LDPC codes is the convolutional variant, which was introduced in 1999 by Felström and Zigangarov [3]. LDPC-CCs are similar to LDPC-BCs in that they are also linear codes that generate code-bits based on parity-check operations [38]. On the other hand, they have a few important attributes that clearly distinguish them from LDPC-BCs:

1. Any given code-bit[1], $v(t)$, is generated using only the present and the $m_s \geq 1$ most recent information-bits[2] (i.e. $u(t-\alpha), 0 \leq \alpha \leq m_s$) and the m_s previously generated code-bits (i.e. $v(t-\gamma), 1 \leq \gamma \leq m_s$).

2. Information blocks of varying bit length may be encoded.

[1]Refers only to the generated parity-check bits exiting the encoder.

[2]Refers only to the information bits entering the encoder, i.e. source stream.

3. BER performance is comparable for the two LDPC code classes. However, for similar BER performance, the memory m_s of an LDPC-CC is much smaller than the block size N of an LDPC-BC [3, 39].

For a rate-1/2 code[3], the time-varying binary information sequence of length[4] t

$$\boldsymbol{u}(0,t-1) = [u(0),u(1),\cdots,u(t-1)]$$

is encoded as the binary sequence of length $2t$

$$\boldsymbol{v}(0,t-1) = [v_1(0),v_2(0),v_1(1),v_2(1),\cdots,v_1(t-1),v_2(t-1)]$$

such that

$$\boldsymbol{v}(0,t-1)\boldsymbol{H}^T_{[0,t-1]} = \boldsymbol{0}. \tag{3.1}$$

As a consequence of these constraints, the parity-check matrix, $\boldsymbol{H}^T_{[0,t-1]}$, is lower triangular, and this reduces the encoding latency to zero and simplifies the encoding process. More precisely, the finite corresponding $2t \times t$ section of the transposed, semi-infinite parity-check matrix $\boldsymbol{H}^T$, denoted by $\boldsymbol{H}^T_{[0,t-1]}$, has the form given in (3.2).

$$\boldsymbol{H}^T_{[0,t-1]} = \begin{bmatrix} h_u^{(0)}(0) & h_u^{(1)}(1) & \cdots & h_u^{(m_s)}(\tau-1) & 0 & \cdots & 0 \\ h_v^{(0)}(0) & h_v^{(1)}(1) & \cdots & h_v^{(m_s)}(\tau-1) & 0 & \cdots & 0 \\ 0 & h_u^{(0)}(1) & \cdots & \vdots & h_u^{(m_s)}(\tau) & \cdots & 0 \\ \vdots & h_v^{(0)}(1) & \cdots & & h_v^{(m_s)}(\tau) & \cdots & 0 \\ & 0 & \ddots & & \vdots & & \vdots \\ & & \ddots & & & & 0 \\ & & & & & & h_u^{(m_s)}(t-1) \\ & & & & & & h_v^{(m_s)}(t-1) \\ & & & & & & \vdots \\ & & & & & 0 & h_v^{(0)}(t-1) \end{bmatrix} \tag{3.2}$$

$\boldsymbol{H}^T_{[0,t-1]}$ has a periodic structure with a period of size τ. The elements of $\boldsymbol{H}^T_{[0,t-1]}$ are either 0 or equal to elements in a $(2m_s+2) \times \tau$ binary phase matrix, $\boldsymbol{P}_{[0,\tau-1]}$. The τ columns of $\boldsymbol{P}_{[0,\tau-1]}$ define τ parity-check constraints that are satisfied for τ subsequent code-bits in the coded bit sequence. The last row of the phase matrix, $\boldsymbol{P}$, is constructed to contain only ones to enable direct encoding (i.e. formulation of the code-bit constraints). Thus, $h_v^{(0)}(t) = 1$, $\forall\, t$.

The entries in the n^{th} column of $\boldsymbol{H}^T_{[0,t-1]}$, where $0 \leq n \leq t-1$, are defined by three regions going down the column:

[3]Code rate $R = b/c$ for $b < c$. Each $h^{(i)}(t)$ in (3.2) for $i = 0,\cdots,m_s$ is a $c \times (c-b)$ submatrix [26, 39]. In this thesis, we consider only regular, rate-1/2 LDPC-CCs, where $c = 2$ and $b = 1$. Hence, the rate can be calculated by $R = 1 - \frac{J}{K}$.

[4]Length t is short for t binary symbols in t time units, i.e. one binary symbol per time step.

(a) If $n \leq m_s$, there are no leading elements. If $n > m_s$, then the leading $2(n - m_s)$ elements are all 0. These entries remove from consideration past bits from the coded sequence, $v(0, t-1)$, that exceed the finite memory of the code.

(b) The next $\min(2n+2, 2m_s+2)$ elements are identical to the last $\min(2n+2, 2m_s+2)$ elements in the $(n \mod \tau)$-th column of the phase matrix $\boldsymbol{P}_{[0,\tau-1]}$, respectively. These bits correspond to a parity-check constraint that appears in a correctly coded bit sequence.

(c) The trailing $2((t-1)-n)$ elements are all 0. These entries remove from consideration future bits from the code sequence, $v(0, t-1)$, that cannot have any effect on the parity constraints.

For example, consider the following phase matrix, where $m_s = 1$ and $\tau = 4$

$$\boldsymbol{P}_{[0,3]} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \tag{3.3}$$

The corresponding parity-check matrix $\boldsymbol{H}^T_{[0,5]}$, extended to show six phases, is illustrated in (3.4).

$$\boldsymbol{H}^T_{[0,5]} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

Given the information sequence $u(t) = (1,1,0,1,0,0,\cdots)$ and the example $\boldsymbol{H}^T$ parity-check matrix, the sequence of code-bits can be verified to be as follows

$$\begin{aligned}
v_2(0) &= 0 \cdot u(0) &&= 0, \\
v_2(1) &= 1 \cdot u(1) \oplus 1 \cdot v_2(0) \oplus 0 \cdot u(0) &&= 1, \\
v_2(2) &= 0 \cdot u(2) \oplus 0 \cdot v_2(1) \oplus 1 \cdot u(1) &&= 1, \\
v_2(3) &= 1 \cdot u(3) \oplus 0 \cdot v_2(2) \oplus 1 \cdot u(2) &&= 1, \\
v_2(4) &= 0 \cdot u(4) \oplus 1 \cdot v_2(3) \oplus 0 \cdot u(3) &&= 1, \\
v_2(5) &= 1 \cdot u(5) \oplus 1 \cdot v_2(4) \oplus 0 \cdot u(4) &&= 1.
\end{aligned} \tag{3.5}$$

Once again, the addition operations occur within GF(2) and hence, are XOR operations. The corresponding coded sequence would thus be

$$v(t) \;=\; (1,0,1,1,0,1,1,1,0,1,0,1,\cdots).$$

We verify that the code is regular, which implies that the number, K, of ones in each row of $\boldsymbol{H}$ and the number, J, of ones in each column of $\boldsymbol{H}$ are constant. For a (m_s, J, K) LDPC-CC, parameter m_s determines the amount of memory. This parameter is monotonically related to BER performance, but the associated increase in encoder and decoder complexity are also important considerations [39].

Another important parameter is the repetition period τ of the $\boldsymbol{H}^T$ matrix. It has been shown that increasing τ tends to improve BER performance [40]. However, this performance increase is much less when compared to the effects of increasing m_s. Choosing $\tau > m_s$ adds phases to the decoding process, thus assisting synchronization in the hardware decoder. In this thesis, we restrict ourselves to $\tau = m_s + 1$ phases.

Given an $\boldsymbol{H}^T$ matrix, we can derive parity-check equations to generate the code-bits at the encoder and then reconstruct the original information stream using the received information-bits and code-bits at the decoder. This would mean we have τ phases - and hence, τ different operations for τ subsequent phases, $\phi(t)$ - for the encoder and decoder circuits before the operations repeat themselves.

3.2 Code Construction & Bipartite Representation

LDPC-CCs are an alternative to LDPC-BCs, therefore, they share and differ several properties. However, their operations are mathematically equivalent. With that notion, it is logical to note that a derivation of an LDPC-CC can arise from a LDPC-BC parity-check matrix $\boldsymbol{H}$ and vice versa. There exist several techniques to construct these codes but we unfold the simple steps of the Jiménez-Zigangirov method presented in [3].

Fig. 3.1 illustrates a method of converting from a $(N = 16, J = 3, K = 6)$ block code to a $(m_s = 8, J = 4, K = 8)$ convolutional code. Visually browsing 3.1(a) -

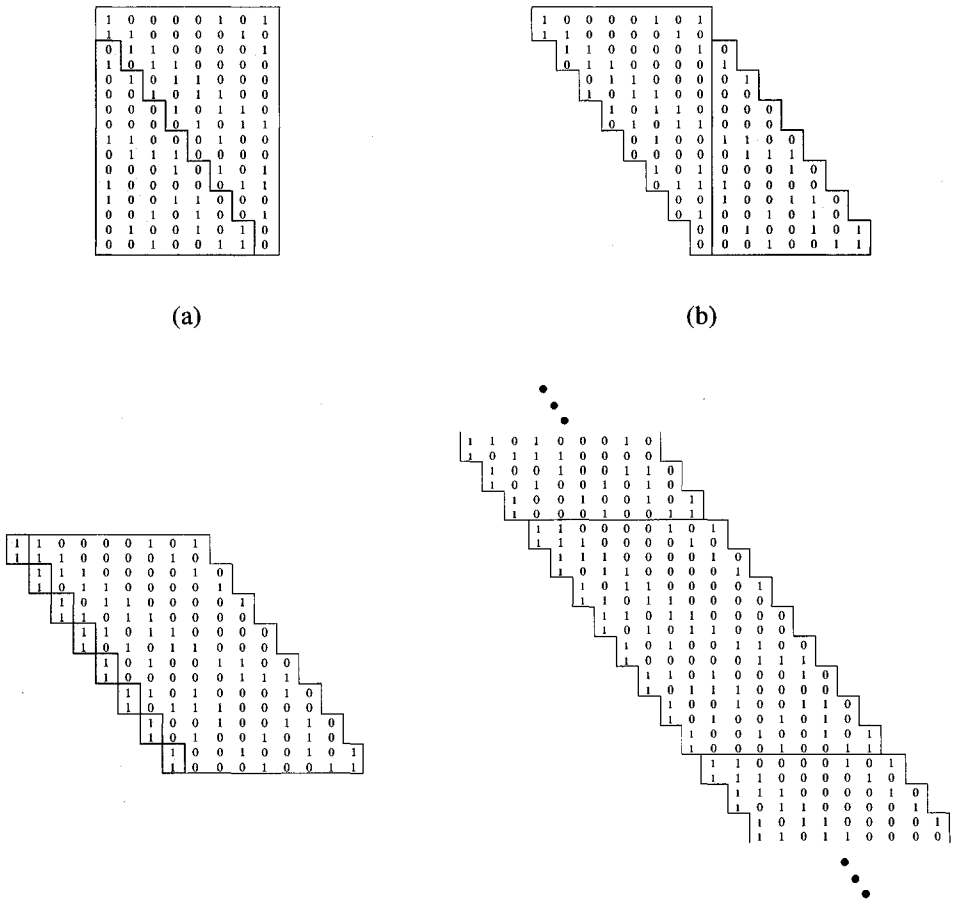


Figure 3.1: Jiménez-Zigangirov method for LDPC-CC construction [3] demonstrated on the $\boldsymbol{H}^T$ matrix seen in (2.16).

3.1(d), we basically transform the transposed $\boldsymbol{H}$ matrix from the LDPC-BC to include the grayed portions one step at a time. This yields the transposed parity-check matrix, $\boldsymbol{H}^T$, for the LDPC-CC that is semi-infinite yet periodic. For a detailed discussion on the construction of LDPC-CCs including other techniques, refer to [3, 26, 40–43].

A portion of the infinite bipartite representation for this $(8, 4, 8)$ LDPC-CC is highlighted in Fig. 3.2. Note that indexing of convolutional codes with respect to time, t, is of importance whereas it is irrelevant for block codes.

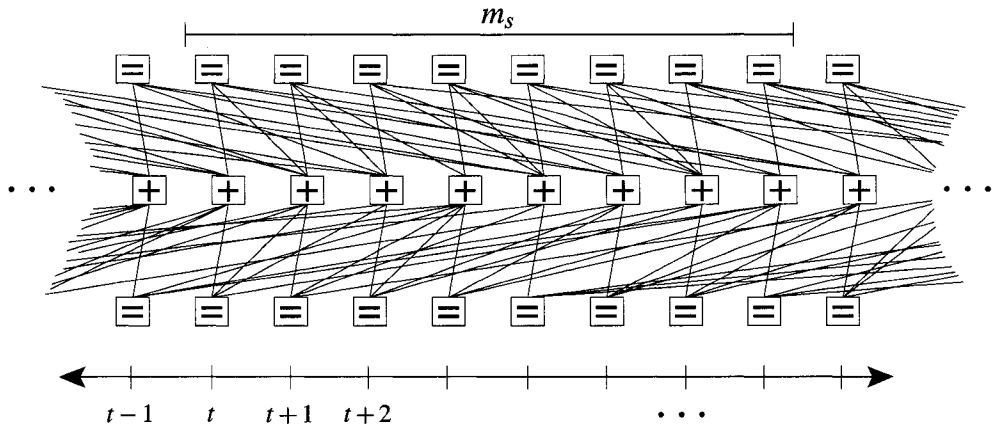


Figure 3.2: Bipartite graph of a $(m_s = 8, J = 4, K = 8)$ LDPC-CC laid out by $\boldsymbol{H}^T$ in Fig. 3.1(d).

3.3 Encoding

Given an information sequence (i.e. $u(0), u(1), \cdots$), we can determine the corresponding encoded sequence (i.e. $v_1(0)$, $v_2(0)$, $v_1(1)$, $v_2(1)$, $\cdots$) for a rate-1/2, systematic and regular code using the equations

$$v_1(t) = u(t) \tag{3.6}$$

and

$$v_2(t) = \sum_{i=0}^{m_s} h_u^{(i)}(t)u(t-i) + \sum_{i=1}^{m_s} h_v^{(i)}(t)v_2(t-i), \tag{3.7}$$

where we assume an initialization state of $v_2(t) = 0$ and $u(t) = 0$, $\forall\, t < 0$. Therefore, the code-bit $v_2(t)$ is computed as a subset of the XOR of the $m_s + 1$ most recent information-bits and the m_s most recent code-bits. The output stream $\boldsymbol{v}(t)$ is obtained by systematically interleaving the stream of information-bits and code-bits.

3.3.1 Termination

LDPC-CCs are great when there is an endless input stream so we can continuously encode, transmit, receive and decode. However, the problem arises when there is a break or time gap in this long sequence, thus creating two separate sequences. This problem is due to the code-bits being dependent on other prior information- and code-bits, i.e. convolutional. Therefore, a "termination sequence" is required and appended to the encoder's output to help bring the encoder back to a known state so as to properly initialize for the next encoding operation. In addition, it sustains the error-correction capability at the decoder for those tail values in the received

message stream. For instance, suppose we have a rate-1/2 encoded frame of size $2n$

$$\boldsymbol{v}(0,t-1) = (v_1(0), v_2(0), v_1(1), v_2(1), \cdots, v_1(n-1), v_2(n-1)), \tag{3.8}$$

where $t = n$ and the termination sequence is to be appended thereafter. We expect the encoder to reach the "all-zero" state – a state where the memory is cleared with zeros – in finite time and hence, the termination sequence cannot just be all zeros. However, there is a period during termination where a zero information-bit input will always yield a zero code-bit output; the point at which this happens is known as the "partial-zero" state [44]. From this point, we need at most m_s information-bits to flush out the encoder but these are not transmitted. Therefore, the rate-1/2 encoded termination sequence is given by

$$\boldsymbol{x}(0,2t-1) = (x(0), x(1), \cdots, x(2L-1)) \tag{3.9}$$

where L is the length of the input information-bit sequence needed to get to the partial-zero state at time $t = 2L-1$. The encoded termination sequence, $\boldsymbol{x}$, can be solved given

$$[\boldsymbol{v}_{1\times 2n}, \boldsymbol{x}_{1\times 2L}, \mathbf{0}_{1\times 2m_s}] \cdot \boldsymbol{H}^T_{[0,n+L+m_s-1]} = \mathbf{0}_{1\times(n+L+m_s)}. \tag{3.10}$$

Fig. 3.3 shows the effect of terminated versus non-terminated frames (or packets) relative to an infinitely long frame (steady state) where termination is not necessary. That is, though the steady state has an infinite frame length, the plot shows a simulated frame length of 2,500 bits to illustrate the effect of terminated and non-terminated frames of finite length. It depicts the relationship between the positions in a frame and the errors encountered. As mentioned before, the reliability of the bits nearing the end of the frame or packet without termination is severed when proper decoding is ceased.

Although the termination problem is neither considered nor a part of this thesis, it is an attribute worth mentioning. Termination is a notable downside for LDPC-CCs and an obvious overhead to the transmitted stream, therefore, work is pursued to minimize its impact and implementation complexity. For more in-depth information, see [44–46].

3.4 Decoding using the MPA

Like LDPC-BCs, LDPC-CCs also employ the same soft-decision, iterative, message-passing decoding algorithm - the MPA. As before, the MPA passes messages containing probabilistic information from one decoding iteration to the next for a total of I iterations. Each probabilistic message is represented as an LLR and as they are processed, their certainty will tend to increase as the decoding algorithm converges to a consistent and corrected coded bitstream. In [3], the decoding algorithm incorporated a serially-concatenated chain of I processors that would be analogous to

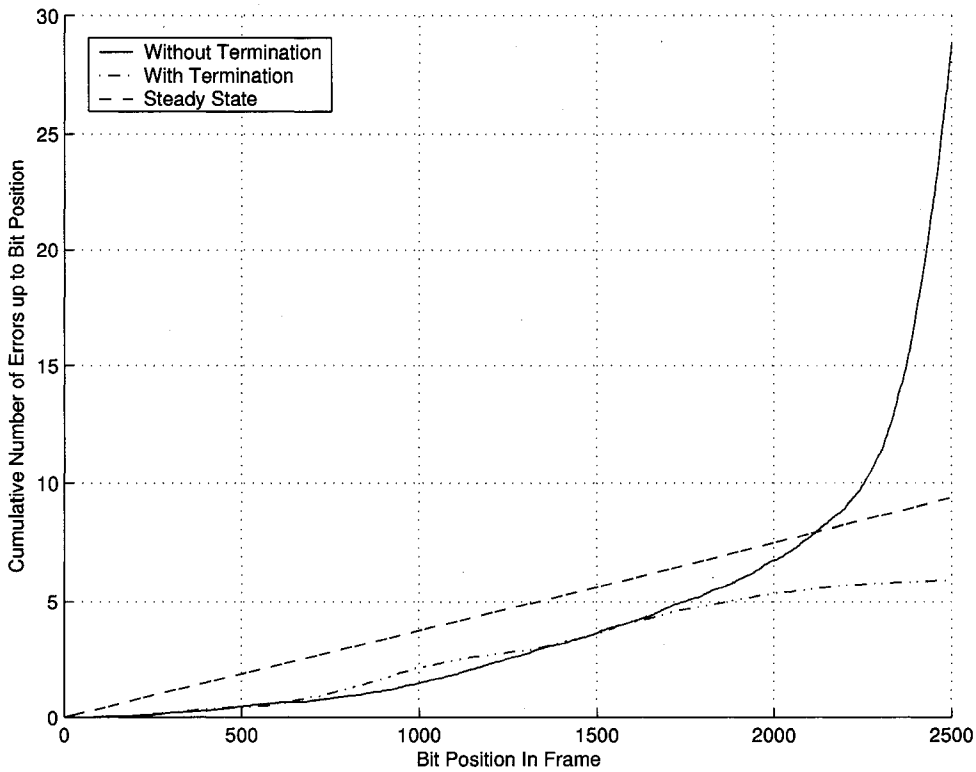


Figure 3.3: Effects of termination on a frame-based (128,3,6) LDPC-CC. *[Graph courtesy of Zhengang Chen, University of Alberta.]*

the I iterations in LDPC-BCs. The $I > 1$ chain of processors all execute the same set of operations at the same time.

Each MPA processor performs the following five steps using $2(J+1)$ first-in first-out (FIFO) delay lines to store the LLRs. Note that Steps 2-5 below apply to every LLR that is shifted in all FIFO delay lines in any processor [3]. Fig. 3.4 shows the LLR data processing steps mentioned below, and Fig. 3.5 shows an abstract datapath of the corresponding decoder.

1) *Initialization*

Initialize all FIFOs in the decoder by loading the dummy belief value "∞" in all locations. Proceed to Step 2.

2) *Shifting Step*

An LLR will be denoted by $\ell_x^{(j)}(t)$, where $j = 0, \cdots, J$ references a particular delay line in either $(x = u)$ the information LLRs, or $(x = v)$ the code LLRs. The index $t = 0, \cdots, \tau$, where $\tau \geq m_s + 1$, represents the minimum latency in clock cycles of a selected delay line. At each synchronization step, we shift in the new information and code LLRs from

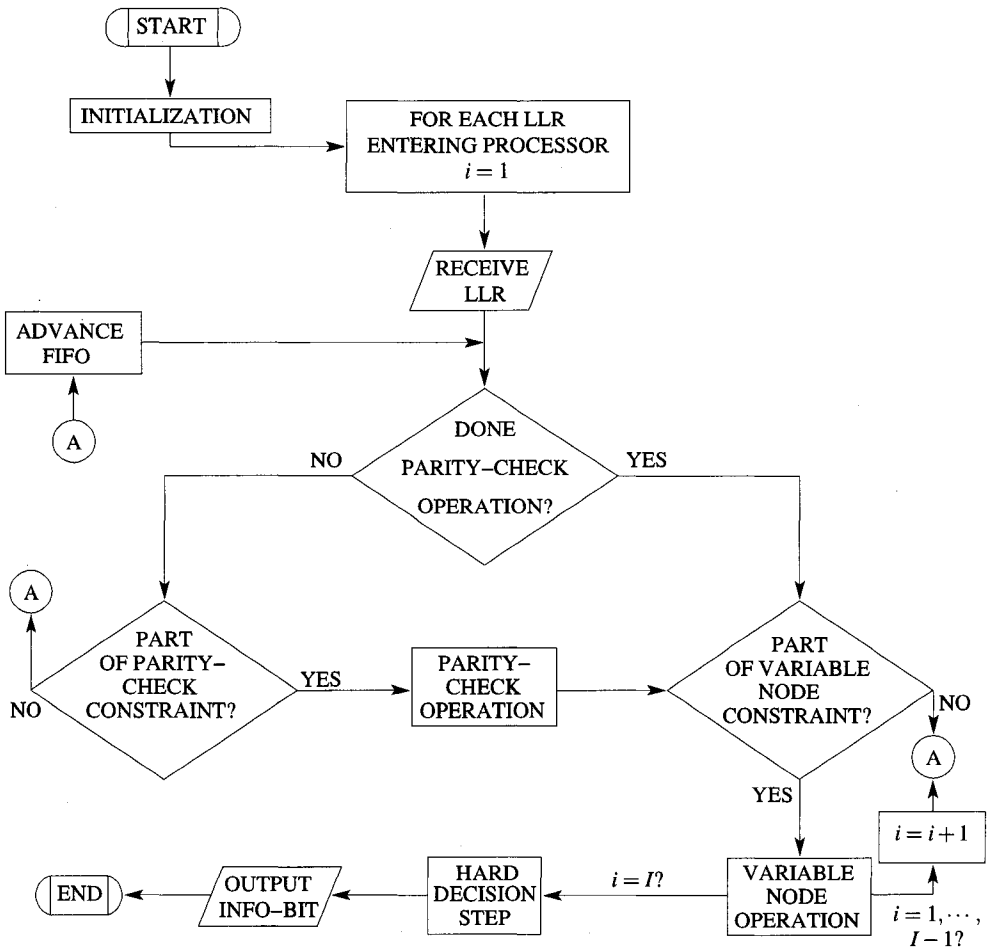


Figure 3.4: Flowchart showing the datapath for an LLR.

the channel - $\ell_u^{(0)}(t)$ and $\ell_v^{(0)}(t)$, respectively - to all the FIFOs in the first processor[5], i.e. $\{\ell_u^{(j)}(0)\} = \ell_u^{(0)}$ and $\{\ell_v^{(j)}(0)\} = \ell_v^{(0)}$ for $j = 0, \cdots, J$. An AWGN channel model with signals modulated using BPSK is shown in Fig. 3.6. These channel LLRs are related to the channel's characteristics by the equations

$$\begin{aligned} \ell_u^{(0)}(t) &= \frac{4KE_b r_u(t)}{JN_o} \\ \ell_v^{(0)}(t) &= \frac{4KE_b r_v(t)}{JN_o} \end{aligned} \tag{3.11}$$

where E_b is the estimated energy per information-bit, $\frac{N_o}{2}$ is the estimated average noise power, J and K come from the (m_s, J, K) definition, and $r_u(t)$ and $r_v(t)$ are the received symbols. The LLRs $\{\ell_u^j(\tau)\}, \{\ell_v^j(\tau)\}$ for $j = 0, \cdots, J$ from processor $i-1$ are input into the FIFOs of processor i.

[5] $\{\cdot\}$ denotes a set of values.

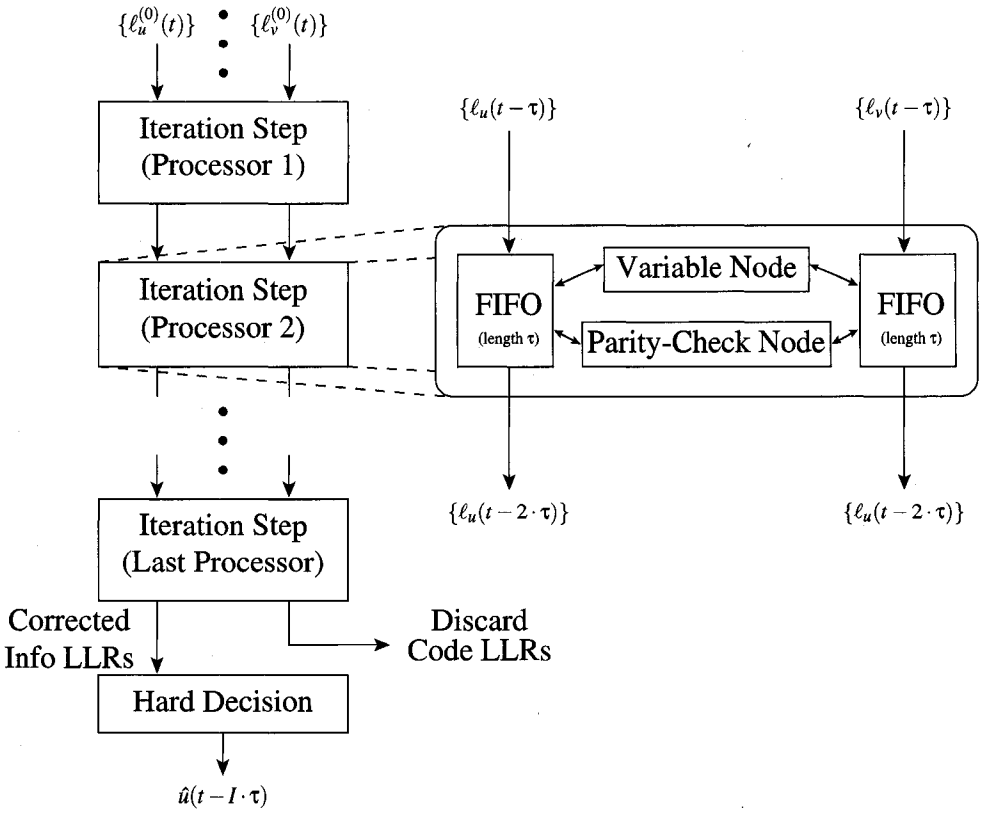


Figure 3.5: Simplified datapath of the decoder.

The LLRs excluding $j = 0$ will be called intermediate as they are altered during the course of τ subsequent phases. If the shifted LLR corresponds to a required parity-check symbol, we proceed to Step 3. However, if the LLR corresponds to a variable node symbol, we proceed to Step 4. Otherwise, Step 2 is repeated.

3) *Parity-check node operation*

In each processor i, $i = 1, \cdots, I$, K intermediate LLRs are read from the FIFOs, altered by the PCN and then the K updated LLRs are written back to their respective positions in the FIFOs. For instance, if one of the K parity-check inputs within a processor i, at some time $t - \delta$, is denoted $\ell_y^{(j)}(t-\delta)$, then $(y, j, \delta) \in C_i(t)$. Here $C_i(t)$ is the set of indices in processor i for all K inputs at each synchronization step, hence $t = 1, \cdots, \tau$. The PCN implements (3.12) where $y, y' \in \{u, v\}$, $j, j' \in \{1, \cdots, J\}$, $\delta, \delta' \in \{0, \cdots, m_s\}$ and $C_i(t) \backslash (y, j, \delta)$ denotes the set $C_i(t)$ excluding the element (y, j, δ). As this operation is quite complex to implement exactly in hardware, a well-known approximation can be employed to reduce the implementation complexity while causing some acceptable degree of perfor-

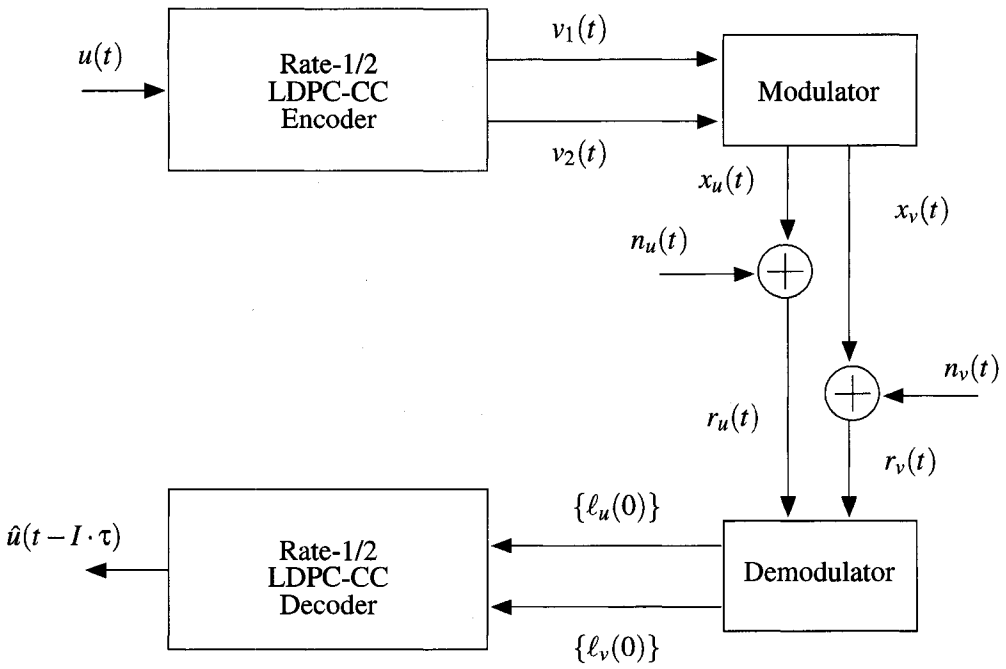


Figure 3.6: LDPC-CC channel model.

$$\ell_y^{(j)}(t-\delta) \equiv 2\tanh^{-1} \prod_{(y',j',\delta')\in C_i(t)\setminus(y,j,\delta)} \tanh\left(\frac{\ell_{y'}^{(j')}(t-\delta')}{2}\right) \tag{3.12}$$

$$\ell_y^{(j)}(t-\delta) = \operatorname{sgn}\left(\prod_{(y',j',\delta')\in C_i(t)\setminus(y,j,\delta)} \ell_{y'}^{(j')}(t-\delta')\right) \min_{(y',j',\delta')\in C_i(t)\setminus(y,j,\delta)} \left|\ell_{y'}^{(j')}(t-\delta')\right| \tag{3.13}$$

mance degradation [47]. The min-sum approximation[6] shown in (3.13) is much easier to realize in hardware. In (3.12) and (3.13), the elements of $C_i(t)$ are determined by the phase, $\phi(t)$, and the locations of the K ones in the relevant columns of the transposed parity-check matrix, $\boldsymbol{H}^T$. If the outputs of the PCN operation are required by the VN operation, go to Step 4; otherwise go to Step 2.

4) *Variable node operation*

The VN operation takes place just before the LLRs leave a processor. This occurs m_s phases after the corresponding LLRs arrive at the processor. In processor i, the VN operation takes J LLRs and computes (3.14), where $y \in \{u,v\}$ and $j' \in \{0,\cdots,J\}$. If the VN is in one of the leading

[6]This same approximation is quite prevalent and is also applicable to LDPC-BCs.

$I-1$ processors, then proceed to Step 2; otherwise, for the I^{th} (i.e. the last) processor, proceed to Step 5.

$$\ell_y^{(j)}(t-m_s) \equiv \begin{cases} \ell_y^{(0)}(t-m_s), & \text{if } j=0 \\ \sum\limits_{j' \neq j} \ell_y^{(j')}(t-m_s), & \text{if } j \in \{1,\cdots,J\} \end{cases} \tag{3.14}$$

5) *Final Hard Decision Step*

The hard decision step takes place for the information-bit LLR outputs of the last processor and determines whether a logic one or logic zero was realized for $\hat{u}(t-I\tau)$ after I iterations. The code-bit LLR outputs from the last processor would be discarded as they encode no useful information. In BPSK modulation as per Fig. 3.6, a logic zero is represented as $x_y(t) = 1$ and a logic one as $x_y(t) = -1$, for $y \in \{u,v\}$. More precisely, the hard decision step is given by

$$\hat{u}(t-I\cdot\tau) = 0.5\left(1-\text{sign}\left(\sum_{j=0}^{J} \ell_u^{(j)}(t-I\cdot\tau)\right)\right). \tag{3.15}$$

In (3.15), sign$(\cdot)$ calculates the overall certainty of the output bit being a one or zero. The remaining events in (3.15) translate the output bit from BPSK to a logic one or zero.

3.4.1 Coding Performance

LDPC-CCs possess the same excellent error-correcting capabilities as their better known block counterparts. In Fig. 3.7, we compare the performance of a rate-1/2 (128,3,6) LDPC-CC with a rate-1/2 (1024,3,6) LDPC-BC. Although the two codes have similar performance, LDPC-CC decoders are easier to pipeline and require fewer parity-check and variable nodes [39]. That is, the small cost of optimizing critical paths of LDPC-CCs is less than for LDPC-BCs.

Both LDPC codes have a regular (3,6) structure and both were simulated with double-precision LLRs, using the approximate min-sum algorithm and sum-product (tanh-based) parity operation. A performance degradation is seen in both min-sum curves.

3.5 Summary

LDPC-CCs offer excellent error performance very similar to their block-code counterpart. They are more advantageous for streaming applications where continuous and simultaneous encoding and decoding operations are of importance. A similar error performance was reported for a (128,3,6) LDPC-CC versus a (1024,3,6)

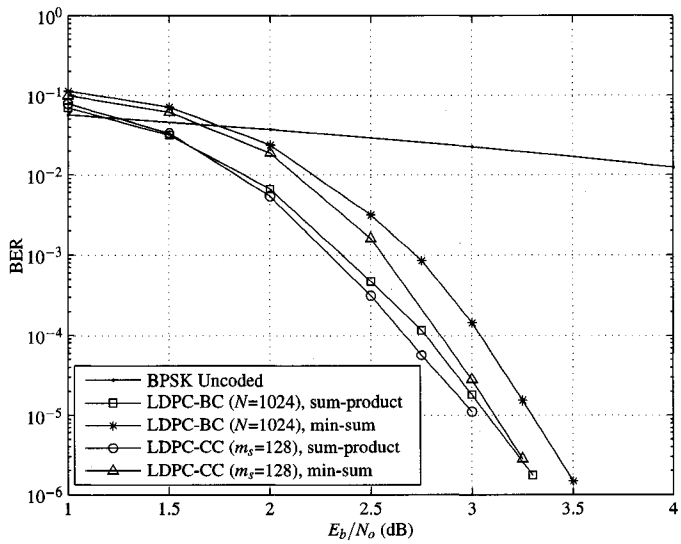


Figure 3.7: LDPC-CCs versus LDPC-BCs with respect to the sum-product and the min-sum algorithms; $m_s = 128$ versus $N = 1024$. *[Graph courtesy of Zhengang Chen, University of Alberta.]*

LDPC-BC. However, their implementation aspects were not discussed entirely. In Chapter 4, we survey hardware implementations noted in industry and academia that showcase LDPC-BC and LDPC-CC systems.

Chapter 4

Prior Work[1]: Hardware Implementations

The demonstrated potential of LDPC codes has led to the design and construction of LDPC-BCs and their implementations in hardware. More specifically, the efficient realization of the decoders are of primary importance as they are the more computationally demanding and complex portions relative to their encoding opposites. Numerous LDPC-BC decoder architectures have been developed in academia and industry [1,48,49]. The recent introduction of LDPC-CCs have also sparked interest and are beginning to make way with their own set of issues and compromises.

In Section 4.1, we briefly summarize the evolution of LDPC-CCs. However, in Sections 4.2.1 and 4.2.2, we elaborate on hardware implementations of LDPC-BCs both from the analog and digital perspectives. The digital section in Section 4.2.2.1 includes a case study of the first noted LDPC-BC implementation.

4.1 LDPC-CCs Since Inception

Much of the work with LDPC-CCs reside in the formation of its theory and comparisons to LDPC-BCs. To name a few, methods of code construction for regular and irregular codes [26,31,32,43], analog decoding techniques [50] and complexity studies [39] have been looked at. Due to the active research of LDPC-BCs, several components are also transferable to LDPC-CCs because of the SPA overlap. Hardware implementations of this variant are still not prevalent but the first among those will be discussed in Chapter 5.

[1]The LDPC-CC hardware implementation described in Chapter 5 is, to our knowledge, the first ASIC implementation of its kind and sent for fabrication in March of 2005. Therefore, this chapter presents a literature review prior to the work in Chapter 5. Since March 2005, several other LDPC-CC architectures have been investigated, but are not mentioned here.

4.2 LDPC-BC Realizations [2]

Since LDPC-BCs were discovered much earlier than LDPC-CCs, physical implementations of LDPC-BCs were and still are more predominant in industry and academia. Architectural developments in this area involve exploring concepts in analog and digital domains as well as improvements in the PCN and VN designs in the decoder component. Several encoder enhancements have also been reported but the cost-benefit gain is greater for the decoder.

4.2.1 Analog decoding

Complementary metal-oxide semiconductor (CMOS) analog decoders have been studied for several years and do foresee several advantages such as smaller circuit sizes, subthreshold operation, reduced routing congestion for the interleaver connecting the node types and much lower power dissipation. The analog principle exploits the exponential properties of transistors to realize the SPA with simple circuits [50, 51].

Fig. 4.1 depicts one method for implementing the basic operations of the SPA. It illustrates what happens with a three-edge node, with ΔV_X and ΔV_Y representing the input LLRs on two of the incoming edges and ΔV_Z being the output LLR on the outgoing edge. V_{bias} exists to bias the circuit and provide a constant current. If analog input currents in the circuit are proportional to the probability masses of the inputs, then by the translinear principle, the differential voltages are LLRs. Only here does the probability and log-domain calculations happen simultaneously, as it becomes only a matter of perspective. It turns out that this one circuit in Fig. 4.1 can perform all the duties of the two node types [4]. The circuit is just replicated but cascaded and wired in a tree format depending on node degrees and code size.

Though there are numerous, unique decoder architectures to outline, almost all of them conceive the following advantages [51]:

1) Potential power savings due to the subthreshold operation of translinear circuits.

2) Smaller node circuits enabling designers to implement larger codes and build for massive parallelism.

3) Continuous-time operation allow output signals to settle unlike the redundant operations seen in the digital implementations. In addition, there is no requirement for a clock network.

4) Reduced routing congestion in the interleaver, often due to how the information is represented. Analog decoders conventionally require only a

[2] Explanations of CMOS theory and transistor operations are not covered in this thesis.

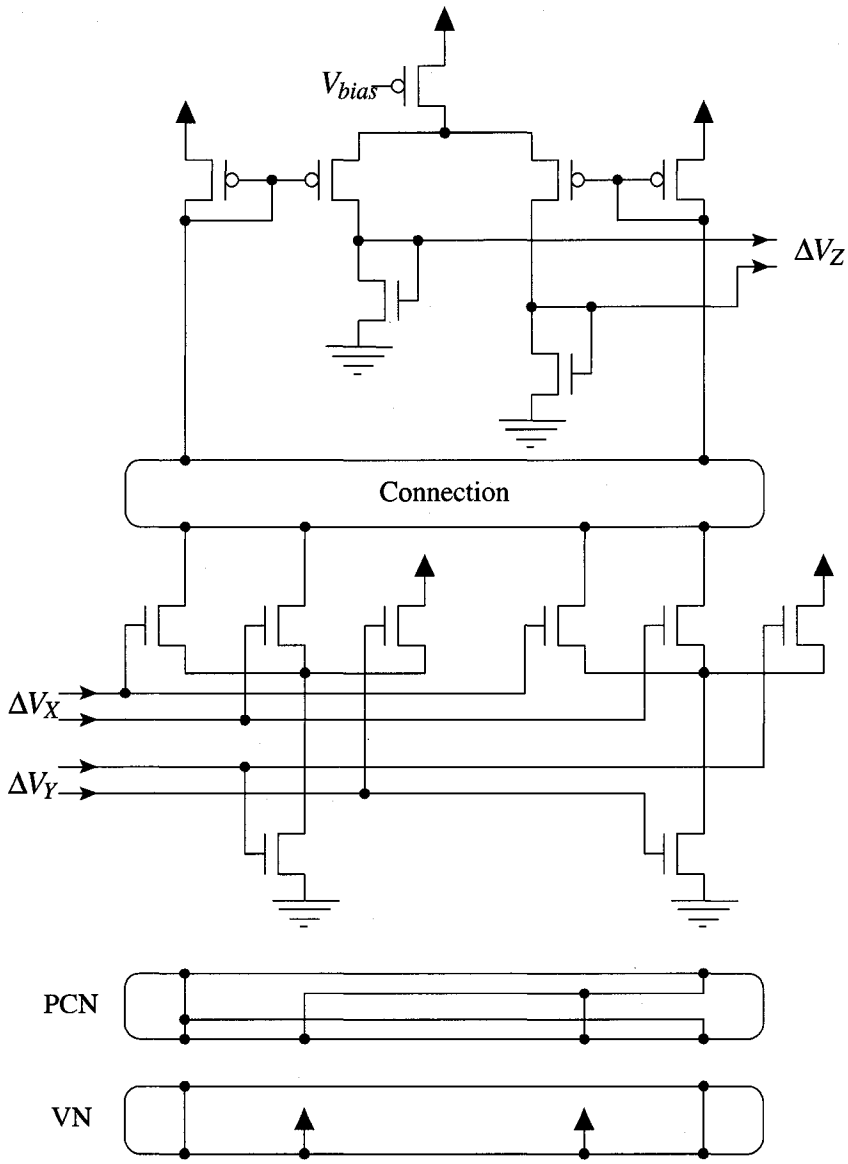


Figure 4.1: A translinear circuit, involving a modified Gilbert multiplier [4], specially made for the SPA [5].

wire or two per message, whereas several wires are need for the digitally quantized messages.

5) Signal representation follows easily from the SPA, where the currents are probability masses and the voltage differences are the LLRs.

6) Straightforward layout synthesis is possible, as analog decoders typically do not deviate from the structure of a bipartite graph. Hence, it becomes easy to comprehend the design.

4.2.2 Digital decoding

Most of the work, like this thesis, focuses on the digital implementation of these decoders. The benefits of choosing digital remain obvious such as resistance to noise, ease of calibration and adjustment and two-mode transistor operation (on and off). However, the main advantage would have to be the capability of attaining complex designs and incorporating large FEC schemes. Large decoding networks, in high performance codes, in analog are quite cumbersome to implement due to problems such as device mismatch and thus, performance is degraded and the risk of circuit failure is increased [51].

4.2.2.1 Howland *et al.*: First Published LDPC-BC Decoder [1]

A rate-1/2, 1024 block-length, irregular code decoder was built with 64 iterations. It features a parallel architecture with soft-decision message passing and a code rate and block size corresponding to a different code type proposed for third generation wireless systems. Fig. 4.2 shows the datapath architecture of their decoder.

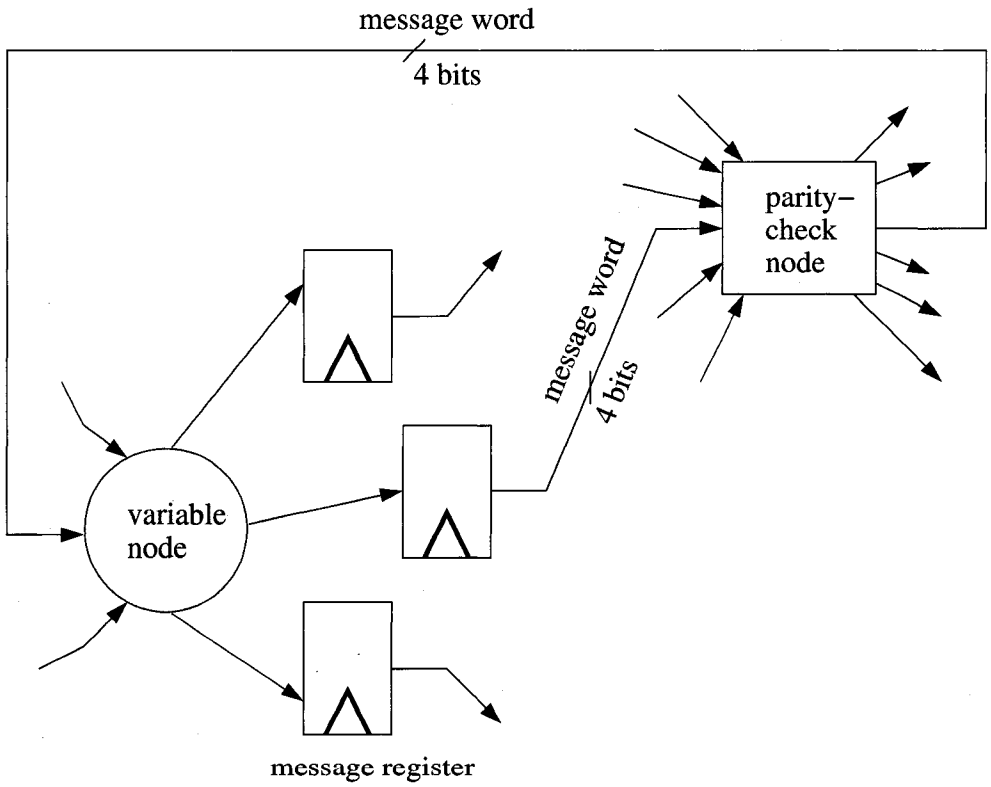
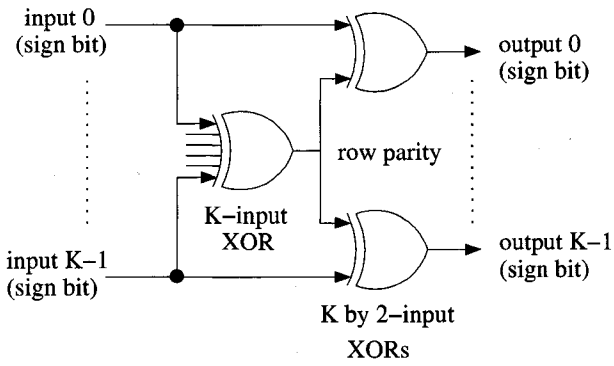


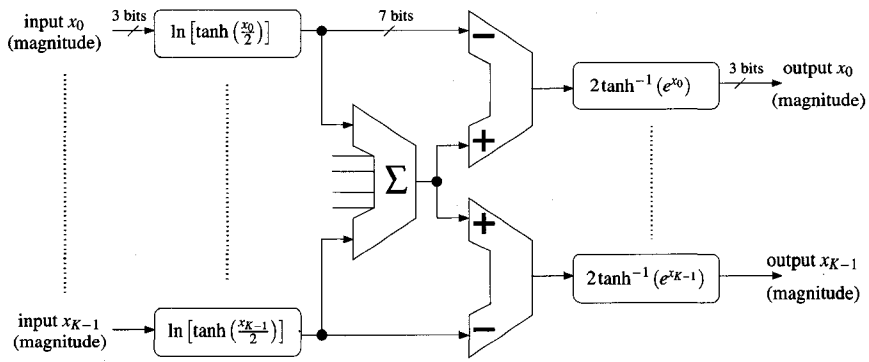
Figure 4.2: Datapath of the decoder in [1].

Contrary to the bipartite graph, their datapath requires two sets of edges: one set for VNs to PCNs and another set in the reverse direction. Though this doubles the number of edges, they eliminate the side effects such as logic overhead, control

signal distribution and use of bidirectional buffers caused by single-set routing. The registers mark the beginning and ending of a decoding iteration and are grouped with the VNs. Thus, the PCNs are fully combinatorial blocks.



(a)



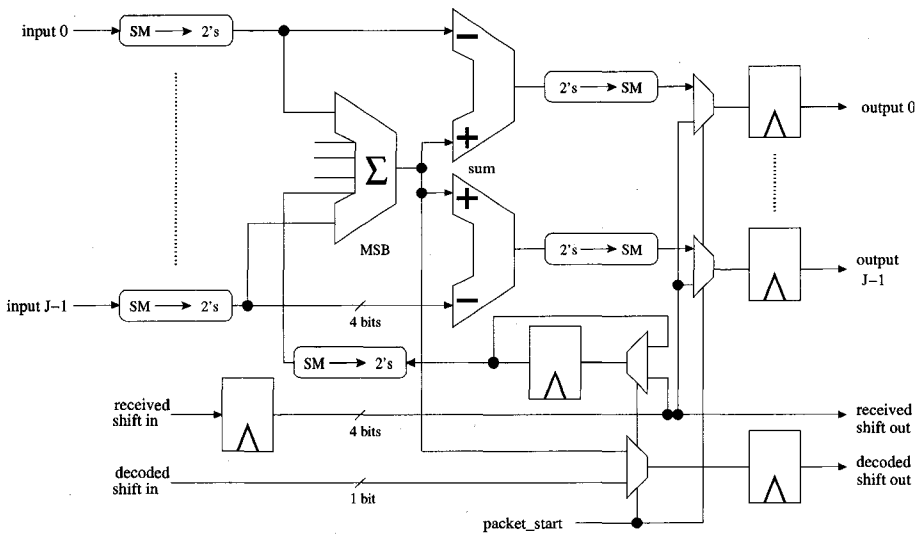
(b)

Figure 4.3: Architecture of the PCN in [1].

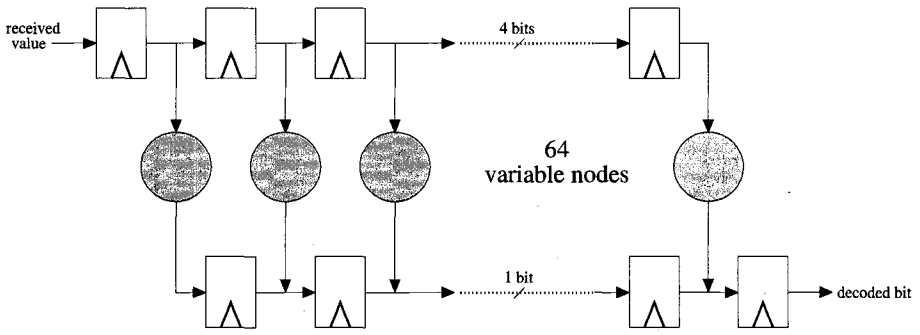
Each LLR message is quantized to four bits in a sign-magnitude fashion – three bits for the magnitude and one for the sign bit. A PCN, as shown in Fig. 4.3, is also split into two separate computations: Fig. 4.3(a) calculates the sign bit and Fig. 4.3(b) calculates the magnitude bits, for all outgoing messages. The magnitude is computed via the sum-product equation in (2.19). The hyperbolic functions dealt with three-bit inputs and outputs so the implementation was straightforward. Internal precision was seven bits for proper overflow management.

The VN architecture is shown in Fig. 4.4(a). The VN performs additions as per (2.20) and hence, conversions from sign-magnitude to two's complement and vice versa are implemented at the input and output interfaces of the node. The sign of the sum of the LLRs is the current estimate of the decoded bit. This can allow them to dynamically vary the number of iterations as this operation is done in every

VN. At the start of a new block, the previous block is loaded into the output shift registers.



(a)



(b)

Figure 4.4: Architecture of the VN and a VN group in [1].

Fig. 4.4(b) shows a VN shift register group for the data input and output scheme. The parallel decoder has a three-block pipeline: as one block is iteratively decoded, the next block is being loaded while the previous block is being read out. Therefore, the register group is replicated 16 times but with different interconnections.

A hardware-sharing architecture using a memory fabric was also described but it proved to be challenging to implement and still achieve the same performance as the parallel scheme. It was concluded that the hardware-sharing architecture required more complicated control logic to oversee memory accesses for LLR messages. Due to the extra periphery logic associated with memory, in conjunction with high bandwidth requirements, it was likely that architecture would consume

significant power and hence, was deemed suitable for low-throughput, area-critical applications.

The parallel scheme provided higher throughput and lower power dissipation for an increase in area. The decoder performed 64 iterations at a clock frequency of 64 MHz to yield a coded throughput of 1 Gbps. Their very large scale integration (VLSI) implementation sized 7.5 mm $\times$ 7.0 mm totaling 52.5 mm^2 for the device area on a 160 nm, 1.5-V CMOS process. The total measured power dissipation was 690 mW at 1.5-V while operating at 64 MHz.

4.3 Summary

The goal of this review was to highlight a selected few of the noted works just to subsequently place LDPC-CCs and its implementation on the same scale as LDPC-BCs. Though there are several commonalities between the two code variations to exploit and optimize, the escape of LDPC-CCs from just theory and simulations must first be established. This becomes more apparent in Chapter 5 as we discuss an encoder and decoder prototype using an LDPC-CC.

Chapter 5

Hardware Implementation of a LDPC-CC

The error-correcting capability of LDPC codes, namely block codes, have uncovered them for potential use in several industrial applications. This has been proven via several software simulations and more recently, hardware implementations in academia and industry. However, the benefits of LDPC-CCs in all physical aspects are still unknown. Appropriate comparisons, such as cost versus performance, design effort, etc., cannot exist until an implementation is developed and its viewpoints are weighted.

We begin by introducing the project's primary intent and then lead into system architecture and ASIC implementation specifics in Sections 5.2 and 5.3, respectively. This includes a discussion of all main components in the communication channel – the encoder, channel emulator and the decoder, along with a mention of peripheral modules. A more recent design involving the use of memory modules is also referenced and evaluated in Section 5.4.

5.1 Project Intentions

The intent of this work (chip designation: ICFAALP2) is to prove feasibility and plausibility of the theory behind LDPC-CCs. Though there exist several architectures for the PCN and VN that could be adopted, we instead provide less complex circuitry so as to lay the groundwork for future enhancements and considerations. It is to be regarded as a prototype for relaying the error-correcting capability of LDPC-CC encoders and decoders. To the author's knowledge, this is the first ASIC architecture showcasing LDPC-CCs.

5.2 ICFAALP2: System Architecture

The LDPC-CC encoder and decoder system was first prototyped on a field programmable gate-array (FPGA) platform using very-high-speed-integrated-circuit hardware description language[‡] (VHDL) and later implemented in Taiwan semiconductor manufacturing corporation's (TSMC) 180 nm CMOS process [52, 53]. This prototype comprises a rate-1/2 (128,3,6) LDPC-CC[†] encoder and a corresponding decoder operating with the min-sum parity-check algorithm, an approximate AWGN channel emulator for built-in-self-test (BIST), test circuitry and peripheral/control modules for performing a number of on-chip operations and controlling data flow.

More specifically, the following are the included modules and their functions:

Test pattern generator: generates pseudorandom bit streams that emulate information streams to the encoder.

Encoder: performs the convolutional encoding of the information stream using a rate-1/2 (128,3,6) LDPC-CC code.

Noise generator: generates and adds an approximate AWGN to the encoded stream to emulate a noisy channel with variable SNRs.

Decoder: comprises 10 serially concatenated processors that carry out the iterative decoding algorithm. LLRs are the inputs and outputs of each processor. Hence, as more processors are added, the BER performance improves.

Slicer: performs the final hard-decision after the last processor and determines the value of the decoded bit.

Error counter: determines the BER using a windowing approach. The values from the test pattern generator are compared to the outputs of the decoder and are validated.

The two main components, namely the encoder and decoder, are placed on the same die to allow for BIST. The BIST is made possible through the use of the test pattern generator, noise generator and error detection circuits. All of the component operations are controlled by an input and output interface, which disperses the user-specified data and control signals to appropriate modules. In addition, the output control block allows us to route LLRs from one processor to any other processor. This becomes important when defects are introduced during the fabrication process, especially since the processors occupy a large portion of the area on the chip. If one link in a serially-concatenated processor chain is broken, the decoder is rendered useless, i.e. almost the entire chip.

5.2.1 Register-based Encoder

The encoding operation described in Section 3.3 can be realized by the circuit given in Fig. 5.1.

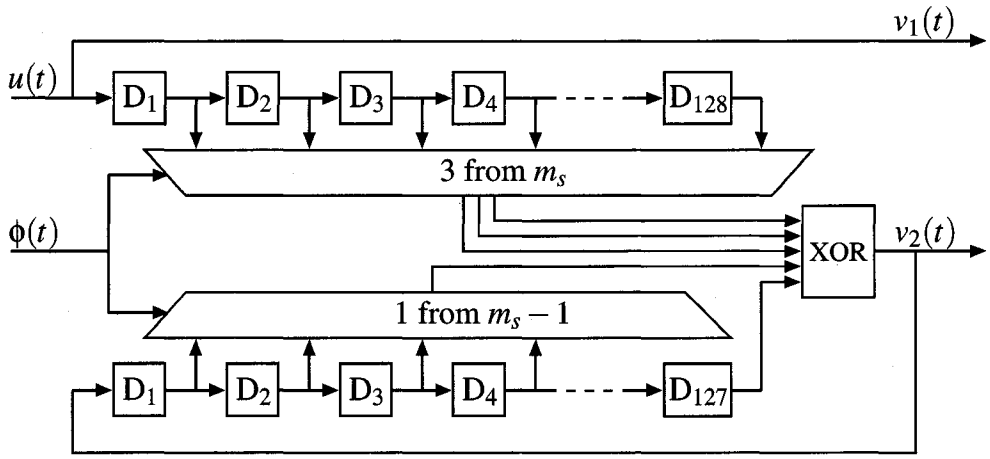


Figure 5.1: Encoder architecture for a rate-1/2 (128,3,6) LDPC-CC.

A direct translation of (3.6) and (3.7) into hardware ideally produce two FIFOs of length τ: one to store the incoming information-bits, $u(t)$, and the other to store the derived code-bits, $v_2(t)$. It should be noted that Fig. 5.1 shows a slightly optimized version of our LDPC-CC code. Specifically, the FIFOs are slightly less than τ since the LLRs near the end are not used and hence, the associated registers can be omitted. The addition operations for $v_2(t)$ are calculated in GF(2) as XOR operations. For our particular LDPC-CC, four of the five inputs to the XOR operation vary at each phase, $\phi(t) = t \mod \tau$. The fifth input, which happens to be a code-bit in our case, is fixed. Two multiplexers, controlled by $\phi(t)$, select the remaining four inputs to the XOR operation: three from the information-bit FIFO and one from the code-bit FIFO.

Unlike LDPC-BCs, data flows unidirectionally through the simple encoder logic. Therefore, there is no need for a control mechanism to read and shift a frame into the encoder, encode the frame, while transmitting the previously encoded frame. As a result, $v_1(t)$ and $v_2(t)$ are produced at the output and forwarded to the channel emulator with very little latency.

5.2.2 Channel Emulator

The simple channel emulator included in this design attempts to model an approximate AWGN channel by exploiting the central limit theorem (CLT) [54]. In this module, the SNR is adjusted by varying the signal energy, E_b, rather than by varying the noise power, N_o. Seven preset levels of signal power are available, ranging from an SNR of -12 dB to 4 dB in steps of approximately 3 dB.

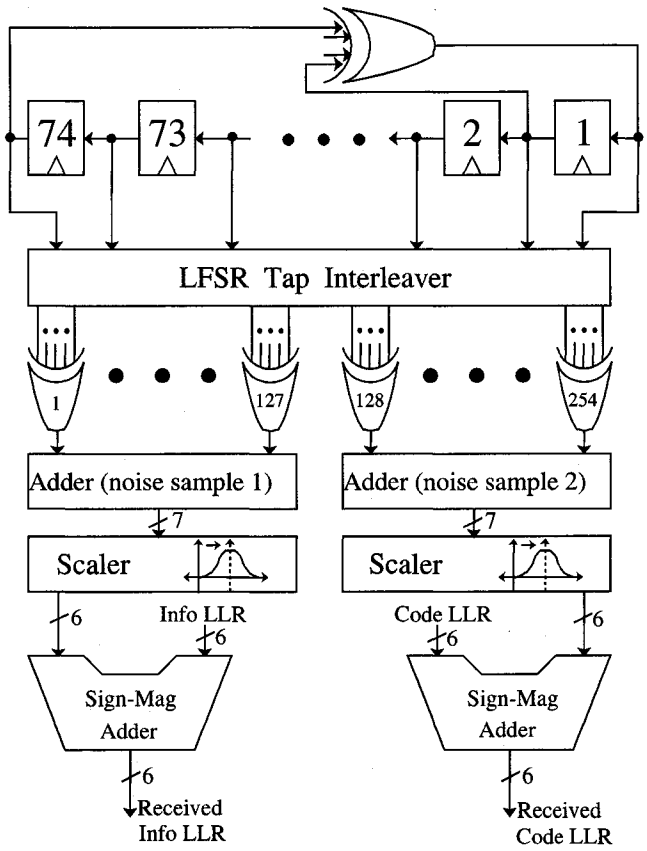
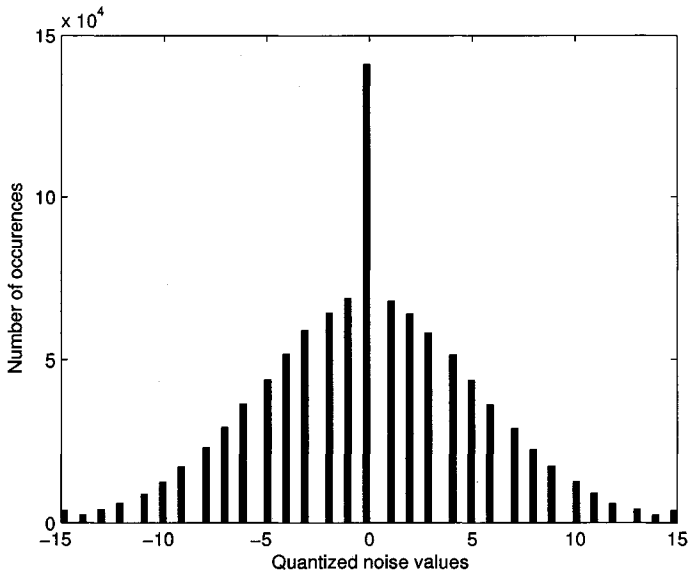


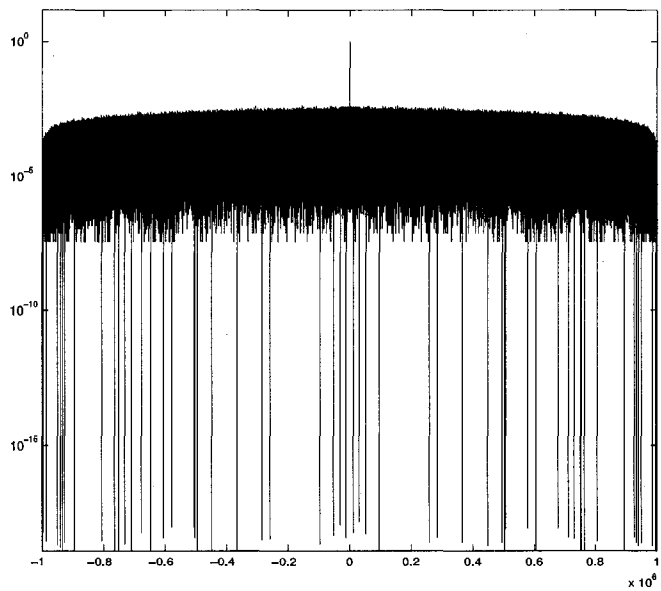
Figure 5.2: An approximate AWGN channel emulator using a 74-bit LFSR as the random number generator.

The emulator, as shown in Fig. 5.2, also converts the input signals to 6-bit LLR values before it enters the decoder. The noise is generated using a 74-bit, primitive polynomial linear feedback shift register (LFSR). For each of the two noise generators, 127 one-bit pseudorandom values are generated by XORing different taps in the LFSR. Making use of the CLT, the 127 one-bit pseudorandom values are then summed to produce a seven-bit noise sample. Thus, one noise sample is available to distort the information-bit and another for the code-bit. This noise generator produces a pseudorandom Gaussian distribution with a mean $\mu = 63$ and a variance $\sigma^2 = 4$. The noise distribution is scaled to be centered around $\mu = 0$ by subtracting 63 from each noise sample. Each noise sample was then quantized to four bits of precision for the magnitude in addition to the sign bit and the saturation bit, which represents the dummy belief value in the initialization step of the MPA.

Fig. 5.3(a) shows a histogram plot after generating $1,000,000$ noise samples and Fig. 5.3(b) shows the autocorrelation function of the same noise samples. It shows that a design error in the noise generator caused the histogram to deviate (a spike around 0) from a Gaussian distribution but the design still produces white (uncorrelated) noise samples; more details are provided in Chapter 6.



(a) Histogram of 1,000,000 quantized noise samples.



(b) Normalized autocorrelation function of the noise samples – absolute value, log-domain plot.

Figure 5.3: Noise characterization.

5.2.3 Register-based Decoder

As shown in Fig. 5.4, the decoder architecture is a cascade of identical soft-decision processors followed by a hard-decision slicer that determines the final decoder output. Belief messages or soft-decision LLRs are represented in the decoder in fixed-point sign plus magnitude format. All nodes in the decoder are initialized to the largest representable positive value in the notation. LLRs estimated from channel measurements are made from an assumed receiver front end. These initial channel LLRs enter processor 1, propagate unidirectionally through the serially-linked chain of processors and exit via the last processor I. At this time, the soft-decision outputs of processor I enter the hard-decision slicer, where associated LLRs are added together according to (3.15). For BPSK emulation, the sign bit of the accumulated result alone determines the estimated information-bit that leaves the decoder.

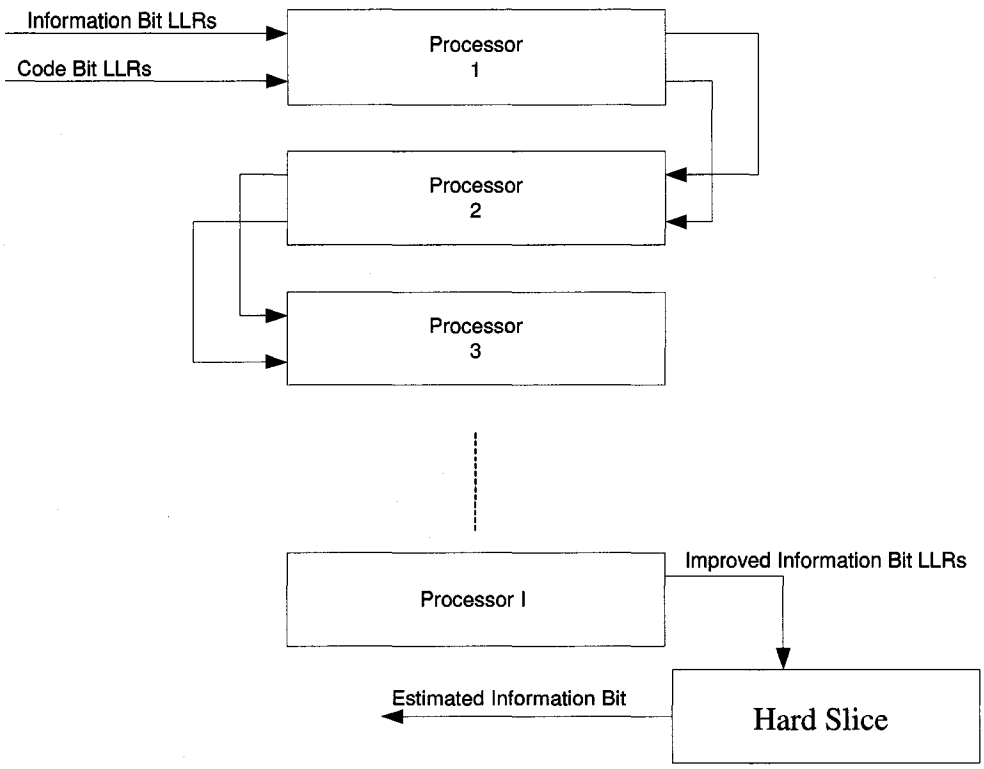
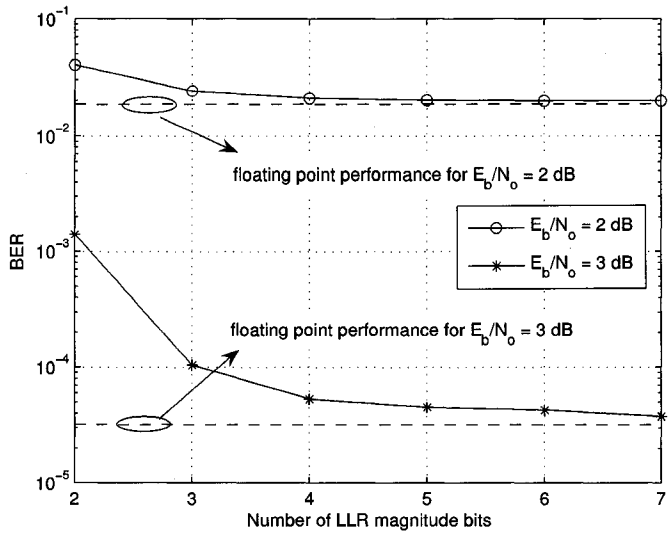


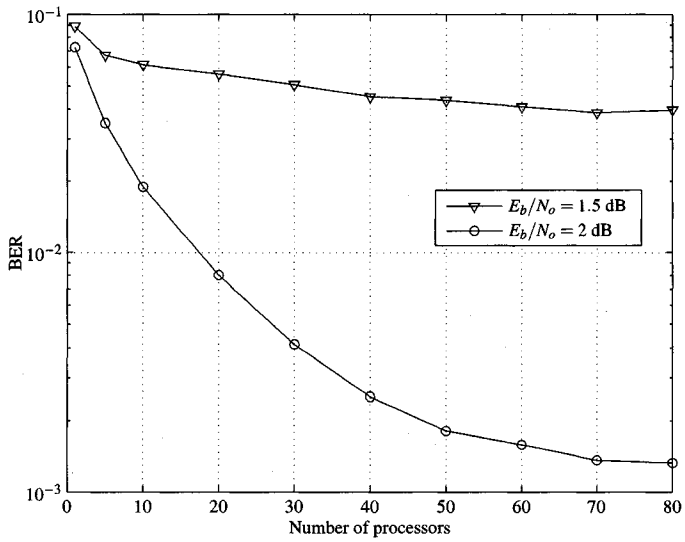
Figure 5.4: Processor chain with I processors and a hard-decision slicer.

5.2.3.1 LLR Precision & Total Processor Count

The major design decisions that affect BER performance and size include the number of required processors and the precision of an LLR (quantization). Software simulations were conducted to determine a suitable value for each design decision.



(a) Plot showing the effect of the number of LLR magnitude bits on BER for a fixed SNR and 10 processors.



(b) Plot showing the dependency of BER on the number of processors or iterations.

Figure 5.5: Rate-1/2 (128,3,6) LDPC-CC design parameters using the min-sum algorithm.

Fig. 5.5(a) shows the dependency of BER on the number of magnitude bits used to represent the LLR. It is clear that the required number of bits varies as a func-

tion of the SNR. If too few bits are used, then the quantization noise becomes the dominating impairment in the decoder. Four bits of precision offers a suitable compromise between complexity and performance for an SNR of 3 dB. In Fig. 5.5(b), we plot simulation results for a varying number of processors. It is clear that at higher SNRs, more processors are required to obtain all the benefits of the error correcting capabilities of the code. We would need around 30 to 50 processors to minimize the errors caused by too few iteration steps. Here the plots show curves for a low SNR and a high SNR, which is used to convey the progressing pattern of the curves with respect to increasing or decreasing SNR. These SNR points were selected as they fall within our test operating region of 1 dB to 3 dB.

5.2.3.2 Processor Architecture

For our rate-1/2 LDPC-CC decoder, each processor contains one PCN and two VNs. The term "node" is used to reflect the analogy between these subsystems in the LDPC-CC decoder and the node processors in an LDPC-BC decoder. Each processor instance in the LDPC-CC decoder is analogous in its role to one iteration in LDPC-BC decoding. For example, BER performance improves as we instantiate more processors until an upper bound is reached [55]. The PCN and VN architectures for a processor are shown in Section 5.2.3.3.

The simple iterative architecture of LDPC-CC decoders makes them ideal for both ASIC and FPGA implementations. The design effort can thus be focussed on the development of one processor. The same processor design can then be tiled as many times as required or as permitted by available resources. A given array of processors can be reconfigured to provide different processing options. For example, we may structure 30 processors as three independent decoders with 10 processors each, or as one large decoder using all 30 processors. The former enables us to achieve higher throughput with multiple data streams at the cost of a lower BER performance in each stream. However, the latter achieves better BER performance for one data stream.

Fig. 5.6 shows a simplified model of a processor. It performs the operations as set out by the parity-check matrix. For each of the $\tau = 129$ phases, specific LLRs are read from the delay lines and forwarded to the parity-check node. The manipulated LLRs are then returned to their respective delay lines. The variable node operation is performed by adders as the LLRs exit the processor, as shown in Fig. 5.6. Each LLR is modified by the parity-check node and the variable node only once in every processor. There are eight delay lines for the storage of the 1032 LLRs, i.e. four dedicated to the information-bits and another four for the code-bits. We note that the processor's latency is greater than τ due to the extra pipelining stages in the parity-check node and the variable nodes. This helped to alleviate some circuit performance bottlenecks but with a slight overhead of some additional control logic.

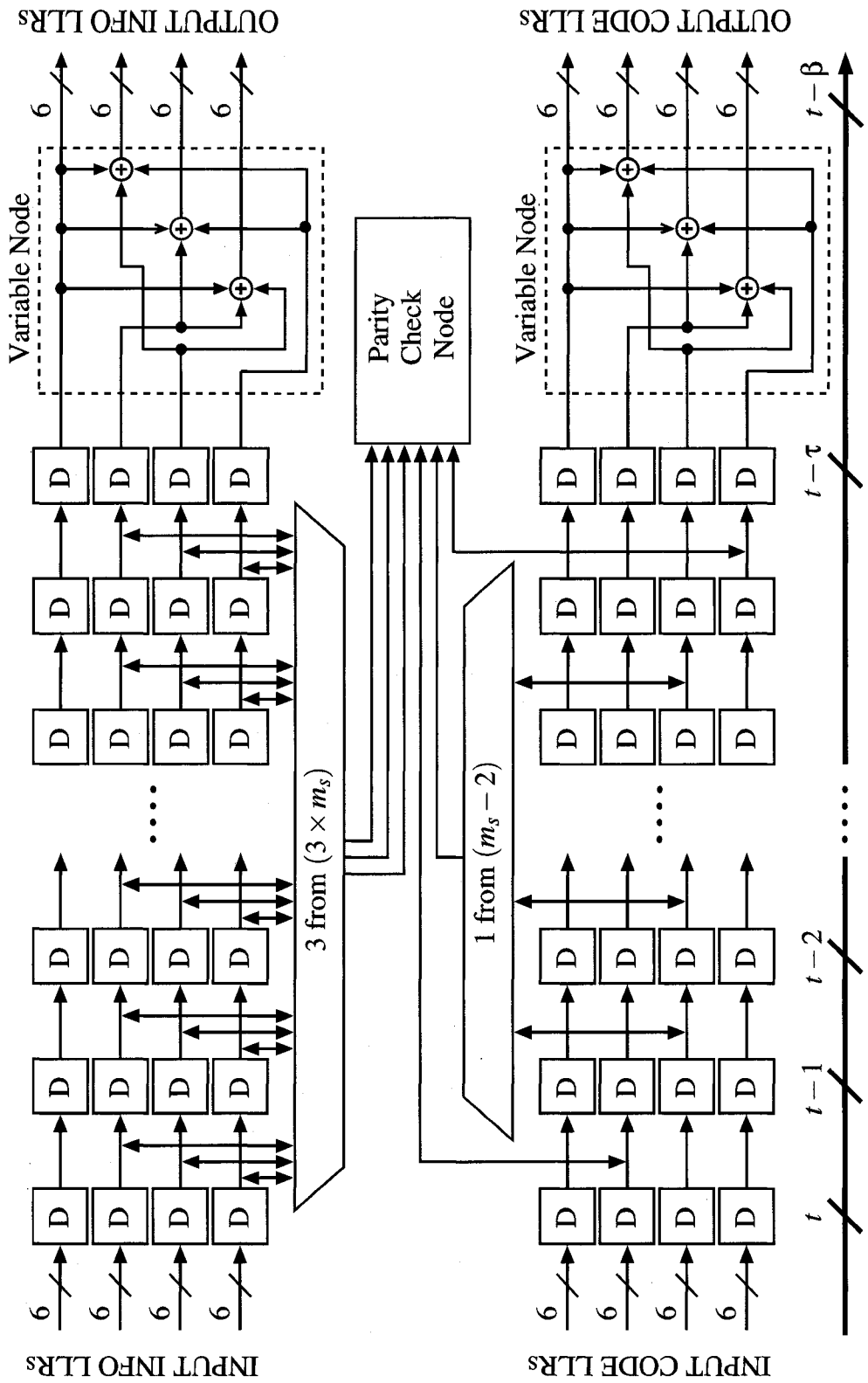


Figure 5.6: Simplified view of a register-based processor where $\beta = 140$ is the latency of each.

Each delay line is six bits wide, corresponding to the width of a single LLR value. Of the six bits, one is allotted for the sign, one bit to indicate saturation or infinity, and four bits for the magnitude or certainty of its received bit. The magnitude bits use a fixed-point representation, where the values range between 0 and 3.75. For simplicity reasons, we will refer to the four-bit values using an integer representation with values ranging from 0 and 15.

Due to limited area on the multi-project silicon wafer, we were only able to incorporate 10 processors in our design. However, it would be straightforward to extend the design with additional processors to achieve better BER performance.

5.2.3.3 Straightforward PCN & VN Construction

The structure of the two node types is very straightforward. For the PCN, the calcuation is broken down into two simultaneous steps: the sign bit and magnitude bits calculations. Fig. 5.7 illustrates a simplified block diagram to calculate the latter. It searches for the first and second minimum magnitudes of the $K = 6$ inputs. Note that are five logic levels, where the first minimum is identified in the third stage and second minimum in the fifth stage using a simple search tree. The sign bit XOR circuitry is rather simple and follows directly from the *sgn* operation in (3.13).

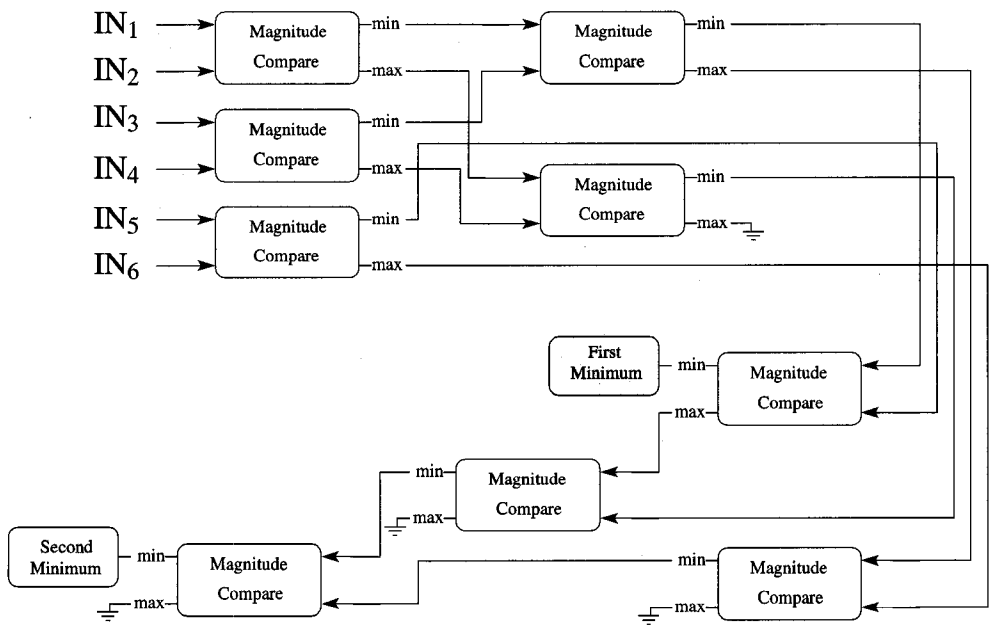


Figure 5.7: Simplified block diagram to implement a portion of the min-sum algorithm. For simplicity, pipeline registers are not shown. Note that unconnected or unused outputs are shown as ground connections.

In a similar manner, the VN performs simultaneous sign-magnitude additions given $J+1$ node inputs, where $J = 3$ in our case. The VN segment in Fig. 5.6 shows

the data flow and is implemented as such. IN0 represents the original channel LLR estimate and is passed unchanged to OUT0. The VN operation follows directly from (3.14) and exactly two of them are situated in every processor.

As these two node types are replicated countless times, it is worthwhile to explore other alternatives to evaluate and minimize the cost incurred.

5.2.4 Peripheral and Control Modules

The peripheral and control modules play an important role in enhancing the testability of the design in terms of functionality and defect avoidance. For BIST, a test pattern generator along with a bit error counter were incorporated. This enabled us to quickly test the chip and obtain BER measurements that confirmed silicon functionality.

Since the ten processors are concatenated, a failure in any one of them could have led to a complete failure of the decoder. To avoid this, we added the ability to inject the channel LLRs into any one of the ten processors. We also ensured that any processor's output LLRs could be connected to the hard-slice unit. Therefore, in the event of any single processor failure, a nine-processor decoder would still be realizable.

5.2.4.1 Test Pattern Generator

The test pattern generator is built with four simple modes of operation due to limited silicon area:

Pseudorandom information-bit generator: generates pseudorandom bits that are input to the encoder. It comprises a 32-bit LFSR with the primitive polynomial

$$x^{32}+x^7+x^6+x^2+1. \tag{5.1}$$

Here, x represents a 32-bit delay line, the exponents are the tap points and "+" is an XOR operation. Given an arbitrary seed for the delay line, the LFSR can generate $2^{32}-1 = 4,294,967,295$ bits before the stream is repeated. This can be used to simulate real-life information bits entering the encoder.

Alternating bit mode: generates a stream of alternating ones and zeros. The circuit is very simple while allowing for quick verification of the decoder's output.

Bit delay mode: includes a set of four registers that initially output ones for the first four clock cycles and zeros thereafter. Again, this is useful for quickly verifying the decoder's output. As the initial latency is long enough, this can help us visually identify the expected output and when it should occur.

All-zero input mode: the simplest mode of all induces the all-zero codeword and thus, responsible for accentuating the effects of noise in the channel emulator.

5.2.4.2 BER Counter

The purpose of this block is to keep track of the decoder's output bits that are in error. The method involves matching the decoder's output with the encoder's input offset by the processor-chain latency. The obvious way would require a buffer (delay line) to hold the inputs until they are ready to be compared and we count until we capture 100 error events (typical case). However, this would be inefficient if the decoder's latency is rather large since the buffer length is dependent on it. Hence, there are other ways to achieve a similar result.

If we are optimistic and we assume the decoder is in working condition, we can compare the decoder's output with an alternating one/zero stream. Then there are two counters, one incrementing only when the compared bits match and the other doing the reverse. Since we assumed the decoder being mostly correct, we can take the lower value of the two counters and call that our error rate. Note that this is not accurate and hence, we must capture several error events to obtain a valid estimate - in our case, we capture 512 to 1024 error events.

The BER is reported via a seven-segment display on an FPGA board or as an 8-bit binary output. This 8-bit value has a 32-value lookup table as shown by Table 5.1. The table reports the approximate range of total bits (log base-2 domain) where the error events have occurred, thus yielding a BER estimate. We require at least 2048 bits to pass before the error count condition is satisfied, i.e. if more than 1024 errors have occurred before 2048 bits have propagated, an emulation error results. The bit count is performed by an arbitrary 42-bit vector and when the maximum is achieved, the counter is reset.

5.3 ICFAALP2: ASIC Design Flow & Analysis

As mentioned earlier, the LDPC-CC encoder and decoder system was first prototyped using behavioral VHDL, verified functionally and then carried through a standard ASIC design flow. The design flow was realized with the assistance of a locally developed tool - HDL2GDS [56].

5.3.1 Synthesis Design Flow

The synthesis flow employed is a common digital design flow and is presented pictorially in Fig. 5.8.

There are four main abstractions in the flow: high-level software modeling, behavioral VHDL modeling, gate-level netlist synthesis and physical synthesis. Most

42-bit vector	8-bit output (seven-segment display)	Code letter
2^{11}	"11111001"	'1'
2^{12}	"10100100"	'2'
2^{13}	"10110000"	'3'
2^{14}	"10011001"	'4'
2^{15}	"10010010"	'5'
2^{16}	"10000010"	'6'
2^{17}	"11111000"	'7'
2^{18}	"10000000"	'8'
2^{19}	"10010000"	'9'
2^{20}	"10001000"	'A'
2^{21}	"10000011"	'B'
2^{22}	"11000110"	'C'
2^{23}	"10100001"	'D'
2^{24}	"10000110"	'E'
2^{25}	"10001110"	'F'
2^{26}	"01000000"	'0.'
2^{27}	"01111001"	'1.'
2^{28}	"00100100"	'2.'
2^{29}	"00110000"	'3.'
2^{30}	"00011001"	'4.'
2^{31}	"00010010"	'5.'
2^{32}	"00000010"	'6.'
2^{33}	"01111000"	'7.'
2^{34}	"00000000"	'8.'
2^{35}	"00010000"	'9.'
2^{36}	"00001000"	'A.'
2^{37}	"00000011"	'B.'
2^{38}	"01000110"	'C.'
2^{39}	"00100001"	'D.'
2^{40}	"00000110"	'E.'
2^{41}	"00001110"	'F.'
otherwise	"01001111"	Error

Table 5.1: Seven-segment (8-bit output) display lookup table. Shows total bit count during error capture and its display code.

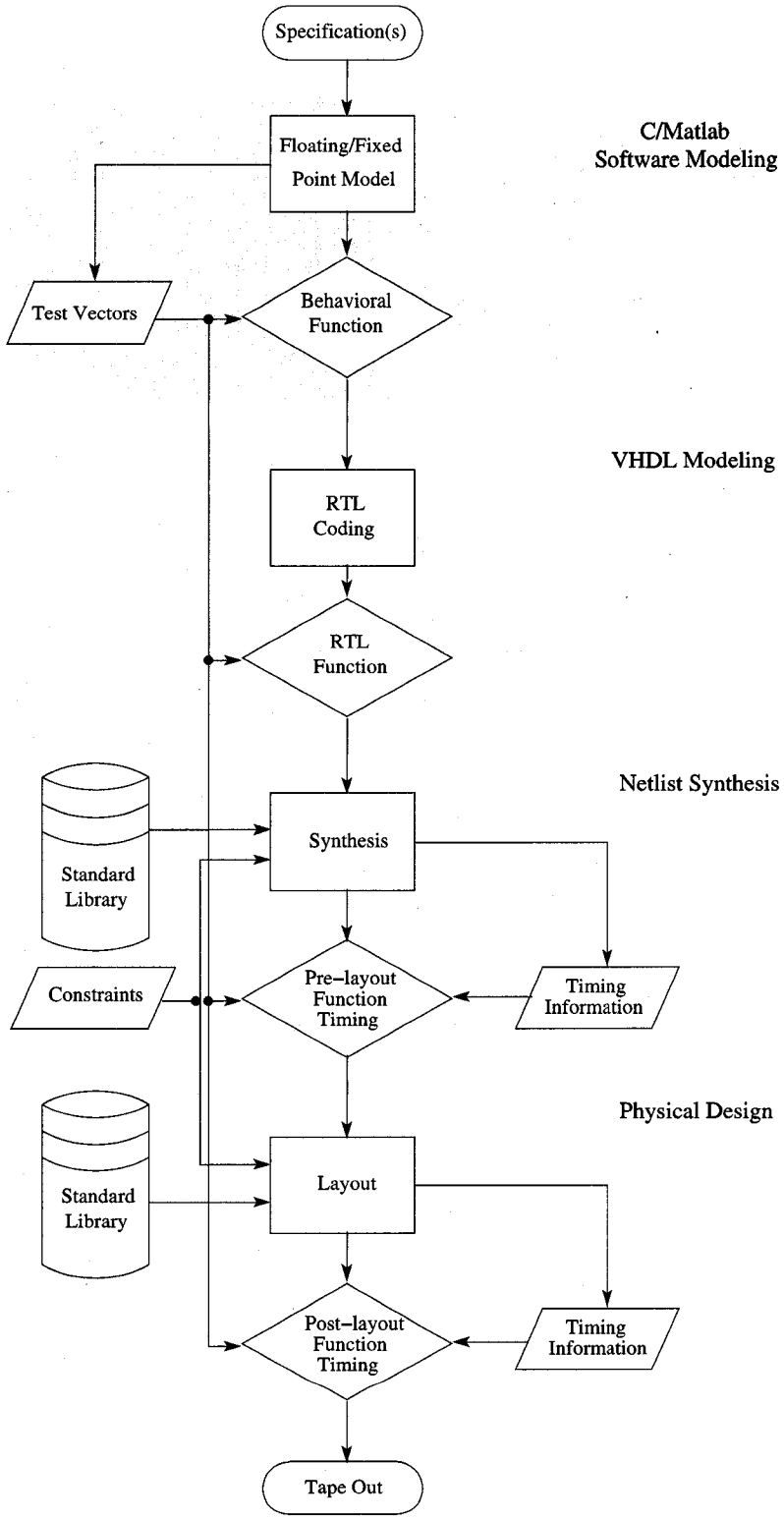


Figure 5.8: Common digital ASIC design flow.

of the effort was spent on modeling, functional verification and netlist synthesis.

The VHDL modeling phase was completed with pre-synthesis simulations to verify that the register-transfer level (RTL) abstraction fully provides the desired functionality. The test vectors employed during early simulations also provide the coverage necessary to satisfy design specifications.

The synthesis phase involves a synthesis tool to:

1) Translate the design description to register elements and combinational logic.

2) Optimize the combinational logic by minimizing, flattening and factoring the resulting logic.

3) Translate the optimized logic level description to a gate-level netlist using only cells from the specified technology library.

4) Optimize the gate-level netlist using cell substitution to meet the specified area and timing constraints.

5) Produce another gate-level netlist of the optimized circuit with accurate cell timing information.

Briefly, the physical design phase involves floor-planning, place and route, layout verification, design rule checks (DRC), layout-versus-schematic (LVS) checks and static timing analysis given a gate-level netlist and other constraints as input. This phase finishes with a post-layout simulation that verifies the desired functionality and appropriate timing requirements.

5.3.2 Tool Set

Several tools were invoked to accomplish the different and necessary tasks. Table 5.2 shows the tool name, along with its function and version.

Additional information can be obtained from [56, 57] and/or by contacting the VLSI design lab at the University of Alberta.

5.3.3 Design Considerations

There were several decisions that made this design successful. More notably, testability was assigned a higher priority over others. As the chip was quite large (also discussed in Chapter 6) with limited BIST features, easing physical testing was a primary concern. To save time and effort, the chip was made to be pin compatible with a prior LDPC-BC decoder design [58] – for input/output (I/O) and power pins.

Function	Tool Name	Version
Software Modeling	gcc C compiler	–
	Matlab	–
VHDL Modeling	*Xilinx* ISE	7.0+
	Mentor Graphics ModelSim	MXE-III
Netlist Synthesis	*Synopsys* Design Compiler	2003.06
Physical Design	*Cadence* First Encounter (place & route)	2003a
	Cadence IC design (layout editor)	2003a
	Synopsys PrimeTime (static timing analyzer)	2003.03
	Mentor Graphics Calibre (DRC & LVS)	2004.2

Table 5.2: List of electronic design automation (EDA) tools required to successfully generate a mask layout.

5.3.3.1 Placement of I/O & Power Pins

Fig. 5.9 shows the bonding diagram for the chip. It is encased in a 120-pin ceramic quad flat-pack (CQFP) package, where 17 pins are left unused and marked accordingly with an 'x'.

Figure 5.9: Bonding diagram for the proposed LDPC-CC encoder and decoder chip. The unused pins are marked with an 'x'.

Table 5.3 shows a list of all pins on the package versus its VHDL reference and/or function. If a pin is a power pin, it can either supply power to the I/O drivers (also known as "ring" power) or to the chip's core (also known as "core" power).

As the design is core-limited, we provide ample power pins. One suggested rule

Pin #	Pin Type	VHDL Reference
1	NC	–
2	OUT	slvHighSpeedOutputsTp(5)
3	RING (V_{SS})	–
4	OUT	slvHighSpeedOutputsTp(4)
5	CORE (V_{SS})	–
6	out	slvHighSpeedOutputsTp(3)
7	CORE (V_{DD})	–
8	RING (V_{DD})	–
9	OUT	slvHighSpeedOutputsTp(2)
10	CORE (V_{SS})	–
11	OUT	slvHighSpeedOutputsTp(1)
12	RING (V_{SS})	–
13	CORE (V_{DD})	–
14	OUT	slvHighSpeedOutputsTp(0)
15	IN	rstResetFp
16	CORE (V_{SS})	–
17	RING (V_{DD})	–
18	IN	slvControlFp(17)
19	CORE (V_{DD})	–
20	IN	slvControlFp(16)
21	IN	slvControlFp(15)
22	CORE (V_{SS})	–
23	RING (V_{SS})	–
24	CORE (V_{DD})	–
25	RING (V_{DD})	–
26	IN	slvControlFp(14)
27	IN	slvControlFp(13)
28	NC	–
29	NC	–
30	NC	–
31	NC	–
32	RING (V_{SS})	–
33	RING (V_{DD})	–
34	IN	slvControlFp(12)
35	CORE (V_{SS})	–
36	IN	slvControlFp(11)
37	CORE (V_{DD})	–
38	IN	slvControlFp(10)
39	IN	slvControlFp(9)
40	CORE (V_{SS})	–
41	IN	slvControlFp(8)
42	IN	slvControlFp(7)
43	CORE (V_{DD})	–
44	IN	slvControlFp(6)
45	IN	slvControlFp(5)
46	CORE (V_{SS})	–
47	IN	slvControlFp(4)
48	IN	slvControlFp(3)
49	CORE (V_{DD})	–
50	IN	slvControlFp(2)
51	IN	slvControlFp(1)
52	CORE (V_{SS})	–
53	IN	slvControlFp(0)
54	IN	clkClockFp
55	CORE (V_{DD})	–
56	IN	slvHighSpeedInputsFp(19)
57	IN	slvHighSpeedInputsFp(18)
58	NC	–
59	NC	–
60	NC	–

Pin #	Pin Type	VHDL Reference
61	NC	–
62	NC	–
63	IN	slvHighSpeedInputsFp(17)
64	IN	slvHighSpeedInputsFp(16)
65	CORE (V_{SS})	–
66	IN	slvHighSpeedInputsFp(15)
67	CORE (V_{DD})	–
68	IN	slvHighSpeedInputsFp(14)
69	IN	slvHighSpeedInputsFp(13)
70	CORE (V_{SS})	–
71	IN	slvHighSpeedInputsFp(12)
72	IN	slvHighSpeedInputsFp(11)
73	CORE (V_{DD})	–
74	IN	slvHighSpeedInputsFp(10)
75	IN	slvHighSpeedInputsFp(9)
76	CORE (V_{SS})	–
77	IN	slvHighSpeedInputsFp(8)
78	IN	slvHighSpeedInputsFp(7)
79	CORE (V_{DD})	–
80	IN	slvHighSpeedInputsFp(6)
81	IN	slvHighSpeedInputsFp(5)
82	CORE (V_{SS})	–
83	IN	slvHighSpeedInputsFp(4)
84	CORE (V_{DD})	–
85	IN	slvHighSpeedInputsFp(3)
86	IN	slvHighSpeedInputsFp(2)
87	IN	slvHighSpeedInputsFp(1)
88	IN	slvHighSpeedInputsFp(0)
89	NC	–
90	NC	–
91	NC	–
92	NC	–
93	RING (V_{SS})	–
94	NC	–
95	CORE (V_{SS})	–
96	RING (V_{DD})	–
97	CORE (V_{DD})	–
98	OUT	clkClockTp
99	OUT	slvHighSpeedOutputsTp(15)
100	CORE (V_{SS})	–
101	RING (V_{SS})	–
102	OUT	slvHighSpeedOutputsTp(14)
103	CORE (V_{DD})	–
104	OUT	slvHighSpeedOutputsTp(13)
105	OUT	slvHighSpeedOutputsTp(12)
106	CORE (V_{SS})	–
107	RING (V_{DD})	–
108	OUT	slvHighSpeedOutputsTp(11)
109	CORE (V_{DD})	–
110	OUT	slvHighSpeedOutputsTp(10)
111	RING (V_{SS})	–
112	CORE (V_{SS})	–
113	OUT	slvHighSpeedOutputsTp(9)
114	CORE (V_{DD})	–
115	OUT	slvHighSpeedOutputsTp(8)
116	RING (V_{DD})	–
117	OUT	slvHighSpeedOutputsTp(7)
118	OUT	slvHighSpeedOutputsTp(6)
119	NC	–
120	NC	–

Table 5.3: List of pinouts versus their respective VHDL reference names, if applicable. "NC" denotes a "no connection."

of thumb is to make power pins equal to 25% of signal pins [59]; in our case, we more than satisfy this requirement while still being pin compatible with the LDPC-BC in [58].

5.3.4 Chip Details: Top-level Overview

The system in Fig. 5.10 requires 38 signals, comprising 20 high-speed inputs and 18 low-speed control signals. The high-speed inputs carry incoming LLRs to the processors in the decoder at the same frequency as the target clock frequency. Similarly, there are 16 high-speed output signals for the LLRs exiting the final processor

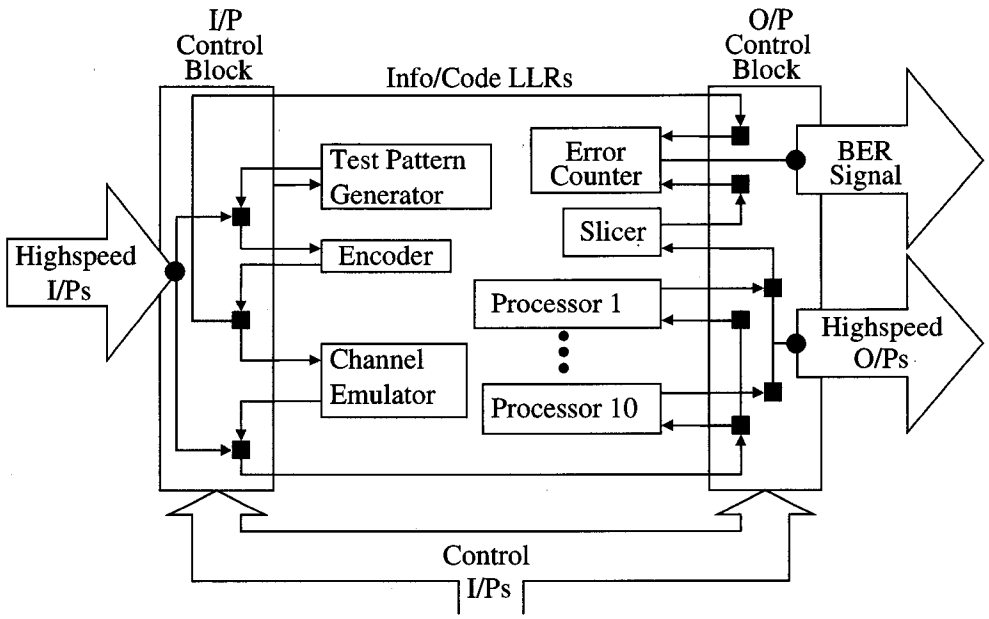


Figure 5.10: Simplified top-level architecture containing an encoder, decoder, channel emulator, BER counter and a hard decision slicer.

Attribute	Encoder	Single Processor	Full Decoder
Area (μm^2)	52588	1006534	10085751
Max. Frequency (MHz)	250	164	164
Power (W)	0.0086	0.39	3.9
Normalized Power ($\frac{mW}{MHz}$)	0.034	3.1	31.2

Table 5.4: Area and power estimates of the (128,3,6) LDPC-CC at the estimated maximum attainable frequency. NOTE: Power estimates for the single processor and the full decoder are reported at a 125 MHz clock frequency. Only 10 processors were synthesized for the decoder.

in the decoder chain and for the cumulative BER up to that time instant. In total, the design contains 103 pins: 57 pins for signal I/Os and 46 pins for power.

5.3.4.1 Initial Synthesis Estimate

We present initial synthesis results obtained using *Design Analyzer*, a tool utilizing version 2003.06 of Synopsys' Design Compiler.

Upon analyzing, elaborating and compiling the designs, constrained by a given clock period, we obtained the area, power and timing estimates.

It is obvious from Table 5.4 that the area and power estimates for the full decoder is approximately $10\times$ that of a single processor. Note the maximum operating frequency for a processor and the full decoder are the same.

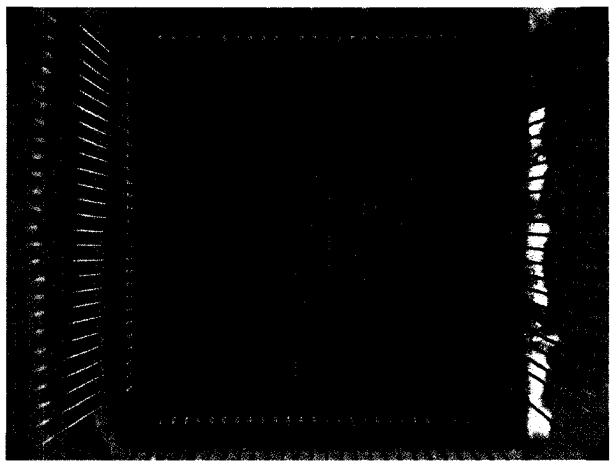
The power figures stated may actually be overly pessimistic because the com-

piler uses only the clock to estimate the switching activity of the circuit. However in reality, the power dissipation should be highest for the first processor and should taper off as we reach the i^{th} processor. That is, the switching activity should be lower in processor $i+1$ than in processor i because the decoder tends to converge to the correct result and therefore, fewer LLR bits need to be switched. This effect is also observed in LDPC-BC decoders [1].

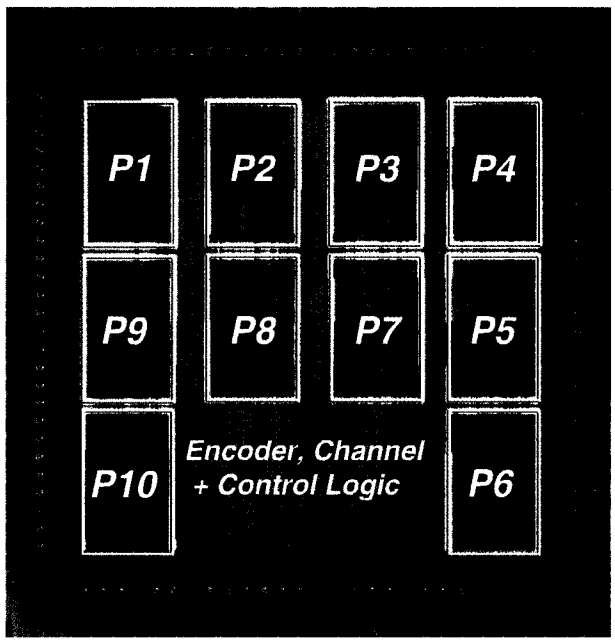
5.3.4.2 Interleaver/Routing Analysis

The 180 nm, 1.8-V CMOS design was specified using behavioral VHDL, then synthesized to a netlist of black-box standard cells and placed/routed using six metal layers thereafter. We employed a two-step synthesis flow, starting with the optimization of a single processor and finishing with the top-level integration of the I/O components, the decoder's ten processors and the remaining modules. Since the number of decoder iterations is analogous to the number of processors, most of the effort can be spent designing a single processor. Therefore, a local synthesis optimization using a bottom-up approach for a processor was performed before processor replication.

From Fig. 5.11, we note that the ten processors that formulate the decoder take up most of the chip's area. A major portion of the processor is dedicated to LLR storage since we require 6 bits/LLR $\times$ 8 rows $\times$ 140 registers/row = 6,720 registers. This circuitry does not include any logic for the parity-check nodes or variable nodes. Another challenge is the number of message nets that are routed to and from the parity-check node. For our particular code, there are 3,060 wires running from the eight register rows to multiplexers, which then select 24 nets to forward to the parity-check node. This directly contrasts the small set of wires entering or leaving a processor. Specifically, 96 wires are required for the input and output of 16 6-bit LLRs and another eight wires carry the control signals. Fig. 5.12 shows a histogram of all the nets and their lengths in our 180 nm, six-metal layer chip for all modules with a bin size of 0.2 mm (lower boundary included) versus the 160 nm, five-metal layer LDPC-BC decoder chip presented in [1]. With over 330,000 nets in our chip, fewer than 0.1% of the nets exceed 0.4 mm; some nets as long as 7 mm could be attributed to V_{DD}/V_{SS} rails, clock tree nets and design-for-testability control circuitry. Also, over 99% of all the nets are 0.2 mm or less, probably due to the simple cascaded processor architecture and the absence of global interleaver wiring. The average net length in Howland and Blanksby's LDPC-BC design was around 3 mm for their top two metal layers, whereas it is only 0.1 mm in our design for all six metal layers [1]. We note that their average net length was obtained before buffer insertion and included nets only in the top two metal levels, inclusive of power/ground nets as well as clock and message nets. Though our average net length calculation was determined after buffer insertion, the number of buffers inserted – 208 buffers on 196 nets – was negligible compared to [1]. In addition,



(a) Original



(b) With overlay

Figure 5.11: Die photograph of the presented LDPC-CC system.

our average calculation includes net lengths after inserting 2,424 clock buffers or inverters. Though this buffer tree is part of our "output" clock pin (for test/debug purposes), it contributes to less than 1% of our total net count and was also seen as a non-factor. Even if the decoder in [1] had six routing layers, a 97% average net length reduction to meet our value would be highly improbable. In the new cascaded processor architecture, the amount of global routing is greatly reduced and

design effort can be focussed on optimizing the cell placement and routing within a processor.

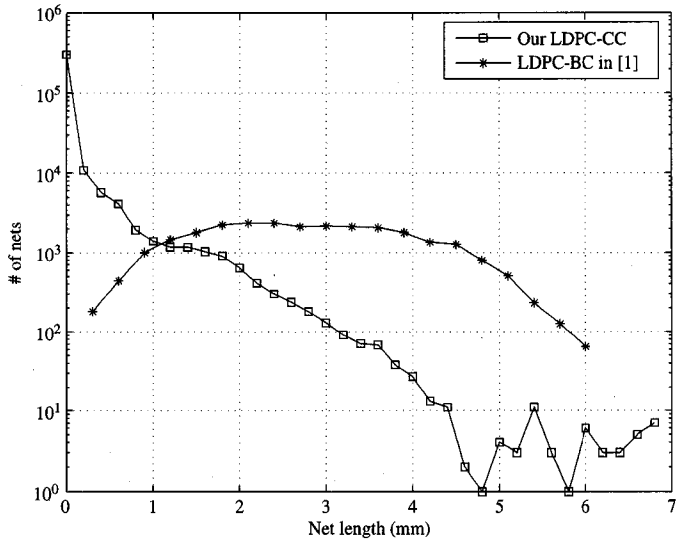


Figure 5.12: Histogram comparison of net lengths in our design versus that of the LDPC-BC decoder in [1].

5.4 Recent Considerations & Alternative Architectures

From recent analysis, a robust improvement over the register-based version is an architecture that utilizes memory. Given that memory and its associated peripherals can be substantially smaller, drastic area reductions can be realized. This is especially true for larger LDPC-CCs where LLR storage density becomes a significant factor and the benefit of using memory is clearly conveyed. However, using memory limits our LLR access bandwidth and thus, complicates the design even further. Here, we summarize a memory-based FPGA architecture presented in [6].

5.4.1 Memory-based, Circular-buffer Processor Architecture

Fig. 5.13 shows a high-level block diagram of an alternative, memory-based processor design synthesized on an Altera Stratix EP1S80 FPGA. The circular buffer is realized using on-chip memory and the current position is tracked using pointers. As multiple read and write operations are required for correct operation, multiple-port memory modules available in modern FPGAs are used. Thus, a dual-port memory is utilized and configured to run at $3\times$ the base clock frequency. This allows for the mandatory seven-read and seven-write operations in the LDPC-CC to complete in one clock cycle.

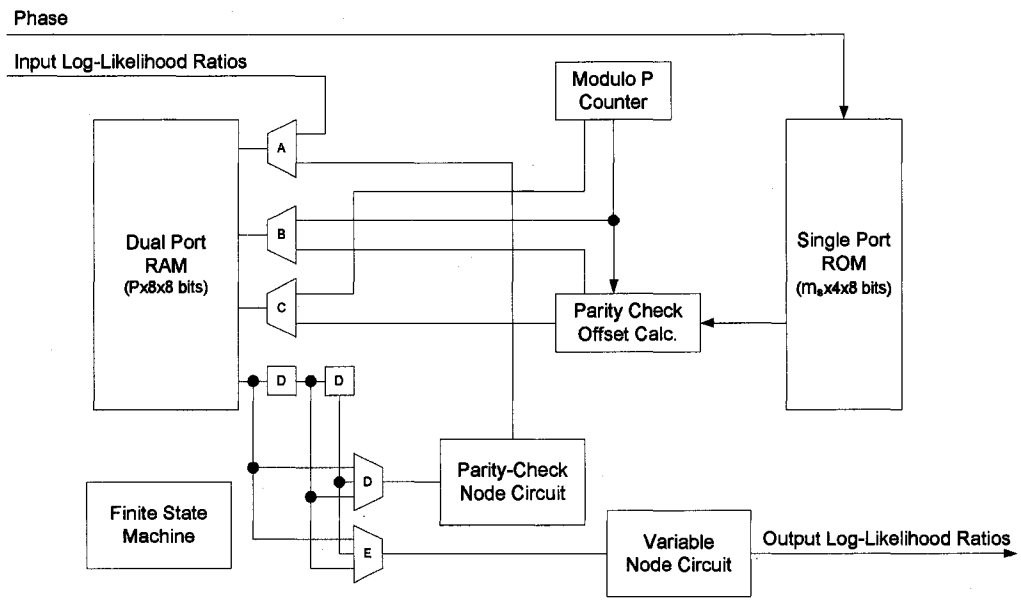


Figure 5.13: The architecture for a memory-based LDPC-CC processor [6]. The gray area represents the portion that is clocked using the 3× clock. NOTE: Only one of the memory blocks need to be configured as RAM; all others as ROM.

A single-port read-only-memory (ROM) is used to store the six offset addresses for each of the $m_s + 1$ different phases. Hence, for a given phase input, the ROM outputs exactly six offsets, of which only four are time-varying. The offsets are used in conjunction with the current pointer location to determine the new read/write address, performed by the parity-check offset calculator in Fig. 5.13.

At any given time, the address to which the input LLRs must be written is generated via a modulo P counter, where P is the processor latency. The outputs of the PCN circuit are then written back to the same addresses in the circular buffer that they were read from. A simple finite state machine (FSM) exists in the shaded area of Fig. 5.13 (3× clock domain) to track the current operation within the random-access-memory (RAM) and ensures that both the input/output LLR operations and parity-check operations are carried out every clock period. This FSM is also responsible for generating all the select signals for the different multiplexers. It should be noted that the logic elements only exist to aid in temporary storage, pipelining and the implementation of logic functions.

This design uses an LLR precision of eight bits and was synthesized on the Altera Stratix EP1S80 FPGA to run at a base clock frequency of 40 MHz. The FPGA version utilizes one-fifth of the resources required by the register-based ASIC processor if it was synthesized on the same FPGA. An in-depth coverage regarding the memory partitions in the single-port RAM is presented in [6].

5.5 Summary

We have shown that it is possible to create a hardware implementation of a communication channel employing LDPC-CCs as the FEC. The choice of a register-based design was arbitrary and adequate to prove feasibility. There have been comparisons of the theory between LDPC-CCs and LDPC-BCs, however, implementation comparisons are still very sparse. It is said that LDPC-CCs can complement LDPC-BCs [39] but more work is required to consolidate the claim. Chapter 6 reports the results for the fabricated ASIC and includes several comparisons[1].

Notes

[‡]Refer to Appendix B for a comprehensive list of all VHDL source files in the ICFAALP2 ASIC project.

[†]The rate-1/2 (128,3,6) LDPC-CC code is intellectual property owned by the University of Notre Dame, Indiana. To obtain the code officially, you may contact Daniel J. Costello Jr. and/or Ali Emre Pusane from the Department of Electrical Engineering via email at *costello.2@nd.edu* and *apusane@nd.edu*, respectively. For reference purposes, Appendix A provides the full (128,3,6) LDPC-CC parity-check matrix file used in this thesis. It also includes a Python-coded parser to convey the file format.

[1]More recent work includes a memory-based, 90 nm CMOS architecture [60]; the work is yet to be published.

Chapter 6

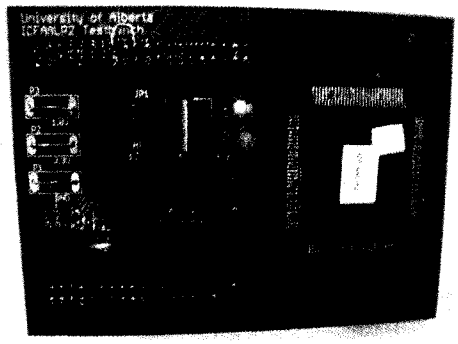
Results

In Chapter 5, we introduced a register-based LDPC-CC ASIC. This design was fabricated via the Canadian microelectronics corporation (CMC) and TSMC. The design included a BIST and other features to aid the testing process. With proper equipment and procedures, we demonstrate a functional encoder/decoder chip and report the necessary results and observations.

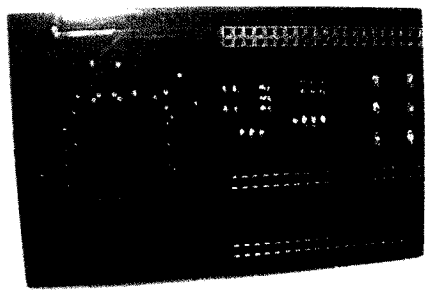
Section 6.1 briefly illustrates the test setup and environment used to carry out the testing process. An explanation of the top-level interface and the supplied input test vectors are detailed in Sections 6.2 and 6.3, respectively. Following it, is an evaluation of the chip in terms of power consumption, area utilization and throughput in Section 6.4. Comparisons against existing work are also provided.

6.1 Test Environment

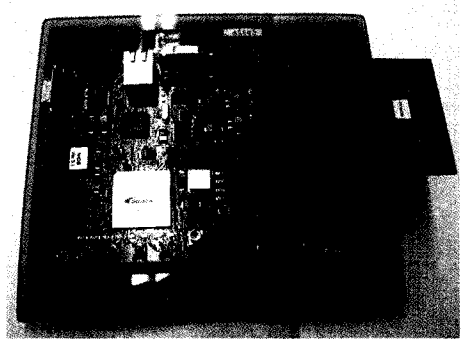
The testing was conducted using three methods: software simulations, ASIC BISTs controlled by an FPGA-based tester and a standard chip tester with a device under test (DUT) and probe station. Software simulations were carried out using a C/-Matlab model. As mentioned before, the 120-CQFP package and placement of the power/ground pins were chosen to comply with a prior LDPC-BC chip to allow the DUT test fixture to be re-used with minimal changes [58]. Since the FPGA testing environment was more versatile, most of the testing was completed with that method – Fig. 6.1 shows the FPGA test setup. This ASIC testbed consisted of a custom printed circuit board (PCB) and an Altera NIOS development board [61]. All the I/O signals on the ASIC pass from the daughter board to the NIOS core, which are then routed to components on the FPGA board or relayed to a software user interface hosted on a personal computer (PC). Once the FPGA has initialized the 18 control bits, all tests were carried out using the on-chip BIST. These 18 control signals are semi-static and hence, issues with glitching or skew were avoided.



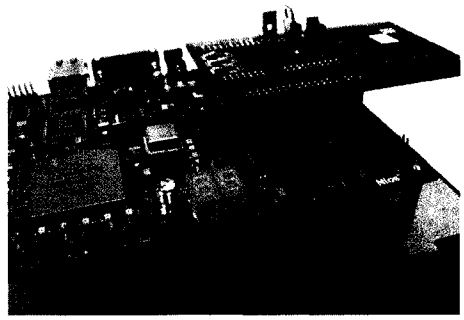
(a) Top side of the DUT (ASIC mounted)



(b) Back side of the DUT (connectors to the FPGA)



(c) Top view of FPGA and daughter board (PCB)



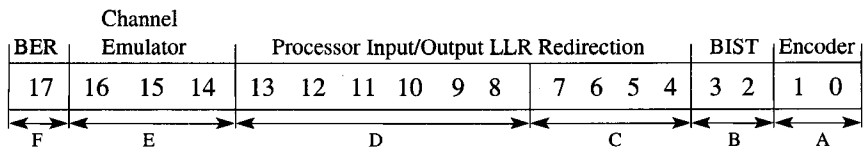
(d) Side view of FPGA and the connected PCB

Figure 6.1: FPGA test setup for the ICFAALP2 ASIC project.

6.2 Top-Level Input/Output Characterization

As mentioned above, there are 18 I/Os that are dedicated control inputs to allow different levels of flexibility. For instance, the 10 processors in theory are supposed to be connected serially. However, for the chip, the surrounding control circuitry defines a multiplexer and switch matrix so we are able to use any processor any number of times and in any order. This feature in particular becomes useful when one of the processors is dysfunctional due to process defects. This way, the entire chip can still be tested but with only nine processors in operation.

Fig. 6.2 shows a high-level mapping of the input control vector, *slvControlInputsFp*. *slvControlInputsFp(1:0)* and *slvControlInputsFp(16:14)* are directly related to the encoder and channel emulator modules, respectively. The channel emulator is controlled only when the encoder is explicitly turned ON. *slvControlInputsFp(1:0)* allow the encoder to operate in three modes: take encoder inputs from



(A) Encoder:
00 = Offchip input data
01 = BIST mode active
10 = Encoder OFF, offchip LLR input data
11 = Encoder OFF, output control vector (test)

(B) Test Pattern Generator:
00 = Pseudorandom (LFSR)
01 = Alternating sequence (010101...)
10 = Delayed sequence (11110000...)

(C) Processor Input Redirect:
Multiplexer to select among the 10 processors. Redirects an LLR into any processor.

(D) Processor Output Redirect:
Multiplexer to select among the 40 pairs of info/code LLRs. Redirects an LLR from each processor's outputs.

(E) Channel Emulator:
000 = No noise
001
°
° = Seven other noise levels
°
111

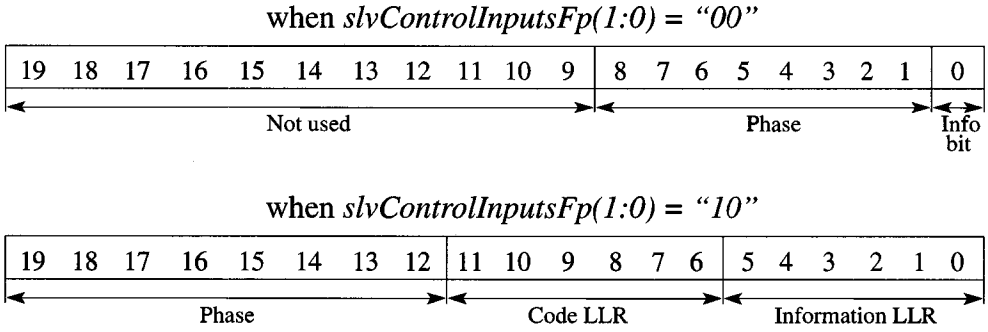
(F) BER Output:
0 = Normal operation, slvHighSpeedOutputsTp(15:0) show LLRs and input/output encoder/decoder bits.
1 = slvHighSpeedOutputsTp(7:0) report BER via seven–segment display.

Figure 6.2: A simplified mapping of the top-level control vector – VHDL reference: *slvControlInputsFp(17:0)*

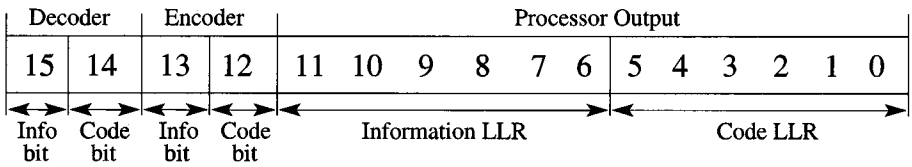
an off-chip source; take encoder inputs from the on-chip test pattern generator (LFSR); and finally, turn the encoder OFF or equivalently, set all LLR inputs to be zero. *slvControlInputsFp(1)*, in particular, is the most important bit as it deciphers how the processors obtain their inputs. *slvControlInputsFp(3:2)* proved useful in all of the initial tests as it is directly related to the BIST and the operation of the LFSR. This allowed us to auto-generate numerous input patterns at core speed and quickly verify system functionality. *slvControlInputsFp(13:4)* illustrates the flexibility and controllability aspects of the design while limiting the total top-level pin count. More specifically, *slvControlInputsFp(13:8)* enabled the user to pull out certain LLRs for intermediate viewing. For our rate-1/2 (128,3,6) code, we were able to view any two LLRs (including channel LLRs) at the current phase and for any of the processor outputs. In combination with *slvControlInputsFp(7:4)*, it was possible to route any processor's output to another processor's input.

Figs. 6.3(a) and 6.3(b) show the mapping for the input and output vectors associated with the datapath. In Fig. 6.3(a), the 20-bit vector is configured differently depending on the encoder's mode of operation. The output vector shown in

Fig. 6.3(b) follows directly from the control vector configuration when the seven-segment BER reading is not used. *slvHighSpeedOutputsTp(15:14)* outputs the hard-decision information-bit and code-bit. *slvHighSpeedOutputsTp(13:12)* shows the encoder information-bit and code-bit outputs. Similarly, *slvHighSpeedOutputsTp(11:6)* and *slvHighSpeedOutputsTp(5:0)* report the pair of information and code LLRs chosen by *slvControlInputsFp(13:8)*.



(a) VHDL reference: *slvHighSpeedInputsFp(19:0)*



(b) VHDL reference: *slvHighSpeedOutputsTp(15:0)*

Figure 6.3: A simplified mapping of the top-level input and output vectors for the datapath.

6.3 Test Vectors & Results

Table 6.1 summarizes the observed results for the different input vectors while operating in BIST mode. It shows the actual input control vector, its expected outcome and whether the decoder passed the test. The hexadecimal vector shown is directly mapped to the control vector (VHDL reference: *slvControlFp*). All of the tests were run at very low frequencies, as the aim was to test for correctness and functionality.

A second set of tests were also run and it involved one test vector to be run at different frequencies. Although this design was synthesized to run with a core clock frequency of 125 MHz, the test results suggest otherwise. The empirical evidence from Table 6.2 suggest repeatable peculiarities at 40, 70 and 80 MHz. It seems that the test results at these core clock frequencies were consistent with se-

slvControlFp(17:0)	Expected	Actual (Pass/Fail)
0x20901	No bit errors with random input	PASS
0x20905	No bit errors with alternating 010101 input	PASS
0x3C901	Many bit errors with random input	PASS
0x3C905	Many bit errors with alternating 010101 input	PASS
0x20001	No bit errors with random input	PASS
0x20005	No bit errors with alternating 010101 input	PASS
0x3C001	Many bit errors with random input	PASS
0x3C005	Many bit errors with alternating 010101 input	PASS
0x2090D	No bit errors with all-zero input	PASS
0x3C90D	Many bit errors with all-zero input	FAIL
0x0AAAA	Expect to see 0xAAAA on output	PASS
0x05555	Expect to see ox5555 on output	FAIL
0x0AAAF	Expect to see 0xAAAF on o/p pins	PASS
0x05553	Expect to see 0x5553 on o/p pins	PASS
0x24905	Some bit errors with alternating 010101 input	PASS
0x28905	Some bit errors with alternating 010101 input	PASS
0x2C905	Some bit errors with alternating 010101 input	PASS
0x30905	Some bit errors with alternating 010101 input	PASS
0x34905	Some bit errors with alternating 010101 input	PASS
0x38905	Some bit errors with alternating 010101 input	PASS
0x2C005	Some bit errors with alternating 010101 input	PASS
0x2C105	Some bit errors with alternating 010101 input	PASS
0x2C205	Some bit errors with alternating 010101 input	PASS
0x2C305	Some bit errors with alternating 010101 input	PASS
0x2C405	Some bit errors with alternating 010101 input	PASS
0x2C505	Some bit errors with alternating 010101 input	PASS
0x2C605	Some bit errors with alternating 010101 input	PASS
0x2C705	Some bit errors with alternating 010101 input	PASS
0x2C805	Some bit errors with alternating 010101 input	PASS
0x2C095	Some bit errors with alternating 010101 input	PASS
0x2C085	Some bit errors with alternating 010101 input	PASS
0x2C075	Some bit errors with alternating 010101 input	PASS
0x2C065	Some bit errors with alternating 010101 input	PASS
0x2C055	Some bit errors with alternating 010101 input	PASS
0x2C045	Some bit errors with alternating 010101 input	PASS
0x2C035	Some bit errors with alternating 010101 input	PASS
0x2C025	Some bit errors with alternating 010101 input	PASS
0x2C015	Some bit errors with alternating 010101 input	PASS

Table 6.1: Test vectors and results in BIST mode.

vere setup and/or hold time violations. The design outputs the received clock signal for test purposes and this output was unstable probably due to the multiplexing, control logic. In fact, instability of this output clock was noticed for every test listed. However, most of them occurred within the first 100 clock cycles and before valid outputs were observed. Due to the lack of detailed test modes, the exact phase of the outputs was sometimes undetermined and even in some cases, the outputs were severely distorted causing numerous bit errors. Hence, it was not always possible to correlate the outputs to its corresponding inputs. In Table 6.2, "GROSS-FAILURE" is used to denote neither a catastrophic hardware failure nor a functional failure. Instead it indicates a problem exists in components other than the decoder since it yields repeatable yet nonsensical data at only certain frequencies. Several iterations of the tests were run and the data obtained each time through was consistent and is reported in Table 6.2.

slvControlFp(17:0)	Operating Frequency (MHz)	Measured Error Count	Actual (Pass/Fail)
0x20905	0.5	0	PASS
0x20905	5	0	PASS
0x20905	7.5	0	PASS
0x20905	10	0	PASS
0x20905	15	0	PASS
0x20905	20	0	PASS
0x20905	30	0	PASS
0x20905	40	GROSS-FAILURE	GROSS-FAILURE
0x20905	45	0	PASS
0x20905	47.5	0	PASS
0x20905	50	0	PASS
0x20905	55	0	PASS
0x20905	60	0	PASS
0x20905	70	117	GROSS-FAILURE
0x20905	80	GROSS-FAILURE	GROSS-FAILURE

Table 6.2: Operating frequency versus error performance (functionality) in a single BIST mode. "GROSS-FAILURE" denotes abnormal operation.

The standard ASIC tester used to run the tests was not properly configured and therefore, the observed bit errors are likely attributed to the skew from the tester. On the other hand, we have also run the tests using the FPGA-based test environment, which shows that the chip can in fact correctly operate at 175 MHz.

6.4 Chip Summary

Average power and BER measurements were obtained from the fabricated ASIC. Table 6.3 summarizes the power readings and other characteristics of the chip while operating at its peak frequency of 175 MHz. Note that although the decoder is the largest and most complicated component, the power figures reported are pessimistic in that they include dissipation from all other modules as well.

Technology	180 nm CMOS 6ML
LDPC-CC Parameters	Rate-1/2, (128,3,6)
Decoder Latency	1,400 clock cycles
Max. Clock Frequency	175 MHz
Max. Information Throughput	175 Mbps
Total Power Dissipation	1.3 W (@ 1.8-V)
Energy per bit	7.6 nJ/decoded-bit
Number of Cells	319 K
Decoder (10 Proc.)	308 K (97%)
Sequential: Storage	67 K (22%)
Sequential: Other	2 K (1%)
Combinational: Logic	239 K (77%))
Other Modules	11 K (3%)
Sequential	1 K (9%)
Combinational	10 K (91%)
Hierarchical Synthesis	Processor level
# of Processors	10
Die Size	3.8 mm $\times$ 3.8 mm
Core Area	9.9 mm^2
Pad Area	4.5 mm^2

Table 6.3: Summary of chip characteristics during peak operation.

6.4.1 Error Correcting Performance

The BER performance of our LDPC-CC is plotted in Fig. 6.4. As mentioned in Chapter 5, a channel emulator was included on the ASIC to assist with the BIST. Although the channel emulator included an approximate AWGN generator, a design error in that block caused the noise to deviate significantly from an ideal Gaussian distribution. In Fig. 6.4(a) we plot the BER performance of the decoder for the seven available signal power settings. It is clear from Fig. 6.4(a) that as the signal power is increased, the error correcting capability improves significantly. It is also clear that using all 10 processors leads to more error correction than using a single processor. This demonstrates that the coding gain of the ASIC increases with the number of processors.

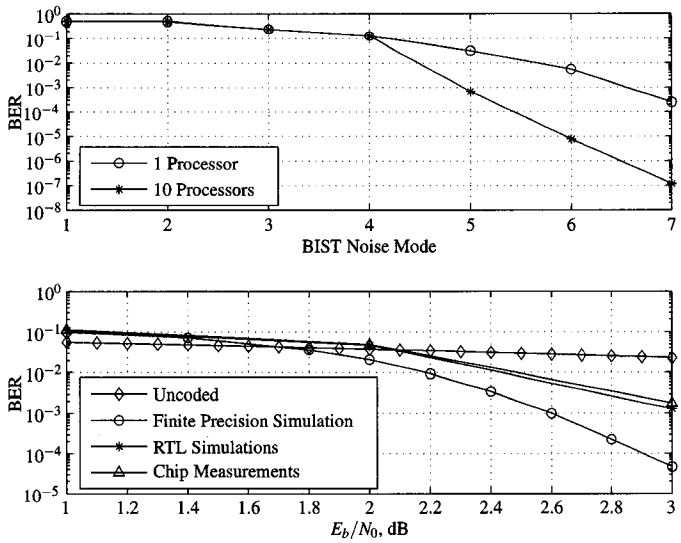


Figure 6.4: (a) The BIST error rate. (b) The simulated and measured BER with AWGN.

We also verified the design functionality and obtained accurate performance results by introducing noise-corrupted LLRs that were generated off-chip and fed into the decoder via the high-speed interface. For the BER data plotted in Fig. 6.4(b), the noise is drawn from an accurate Gaussian distribution. Here the performance of the silicon is compared against a finite-precision software simulation and a VHDL simulation using an event-driven simulator. The VHDL simulation and silicon measurements are in excellent agreement. As expected, the finite-precision simulations give better performance as they do not capture all the saturation effects in the variable node. Again, these results demonstrate that our circuit was properly correcting errors and indeed achieved a BER of 0.0018 (approximately 10^{-3}) at 3 dB. This performance could be improved easily by increasing the number of processors, as shown in Fig. 5.5(b). However, as mentioned in Chapter 5, we were limited to ten processors in the prototype due to silicon area constraints.

6.4.2 Power Analysis in BIST Mode

The power distribution was analyzed under different test modes and for varying supply voltages. The results are plotted in Fig. 6.5. In reset mode, the main source of power dissipation is the clock tree network, which includes all associated clock buffers and related black-box cells. Note that there is no difference in power dissipation in the ten-processor mode versus the one-processor mode and hence, only one curve is shown. This is because an average power measurement was taken from the power pins that supplied the entire digital core and there is no facility to selectively decouple the clock. At a low clock frequency of 1 MHz, we measured 5.4 mW of total power with no inputs/outputs toggling, except for the clock pins.

To estimate the static power from the power results for the reset mode in Fig. 6.5, we used a second order approximation to reduce the residual error. Using this method, the static power consumption was about 3 mW. Fig. 6.5 also shows that approximately two-thirds of the total operating power dissipation is consumed during reset mode, which is due in part to the clock tree network [62].

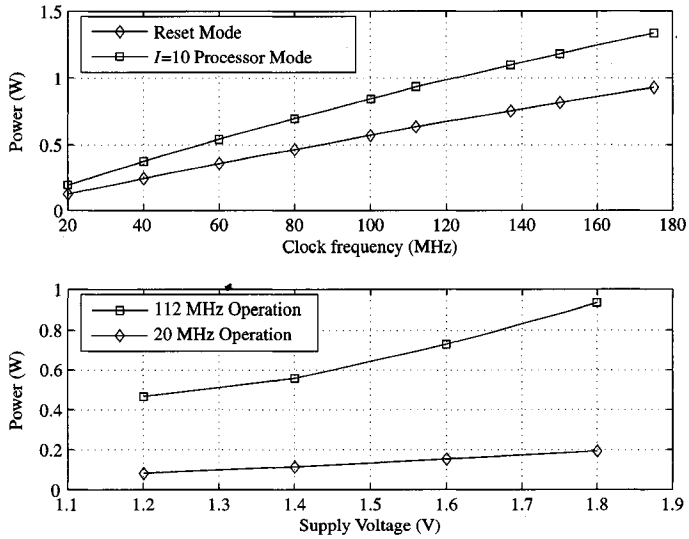


Figure 6.5: Plots show the actual average power dissipation in different modes of operation.

To simplify the prototype's design and reduce design time, the ASIC is a register-intensive design with over 20% of the cells being sequential elements. When this is coupled with the fact that synthesis was time-driven and not power-driven, the resulting energy per decoded-bit was 7.6 nJ at an estimated 3 dB SNR. This is higher than the 1.3 nJ/decoded-bit at 3 dB SNR from Howland and Blanksby's 690 mW LDPC-BC decoder [1].

Using voltage scaling, it is possible to obtain lower power dissipation or higher throughput. In Fig. 6.5, the supply voltage was reduced down to 1.2-V. Here we can operate efficiently at 20 MHz dissipating only 83 mW. This lowers the energy per decoded-bit to 4.1 nJ.

6.4.3 Performance Comparison

As this is the first known ASIC hardware implementation of an LDPC-CC, there are no previous benchmarks to compare against. On the other hand, several LDPC-BC implementations have already been described in industry and academia. Although LDPC-BCs and LDPC-CCs are based on similar theoretical principles, they differ greatly in implementation architecture [39]. LDPC-BC implementations can be fully parallel and hence, much higher throughputs are theoretically possible. LDPC-

Ref.	Code Type	Throughput	Power Dissipation	Technology	Chip Size	Additional Information
[65]	(8176,7156) block, rate=0.875	170 Mbps (*)	–	Xilinx Virtex II FPGA	–	15 iterations
[48]	(9216,3,6) block, rate=0.5	54 Mbps	–	Xilinx Virtex-E FPGA	–	18 iterations, BER=10^{-6} @ SNR=2 dB
[66]	(2048,3,6) block, rate=0.5	640 Mbps	787 mW @ SNR=1.4 dB	180 nm, 1.8-V CMOS	14.3 mm^2	10 iterations, Turbo decoding hybrid, tunable code rate
[67]	(1200,720) block, rate=0.6	461 Mbps (*)	644 mW @ 1.8-V	180 nm, 1.8-V CMOS	25 mm^2	64 iterations using 1.62-V

Table 6.4: Summary of other published digital LDPC code decoder implementations. If and when available, information throughput is indicated by (*).

CCs require significant work to approach the same throughputs. Nevertheless, we will make a comparison with some existing LDPC-BC architectures.

Howland and Blanksby's 1024-b, rate-1/2, parallel LDPC-BC decoder operated at 500 Mbps of decoded throughput using 64 iterations, dissipating 690 mW on a 160 nm, 1.5-V CMOS process [1]. Scaling these results linearly with feature size and quadratically with supply voltage yields an estimated power of 1.1 W in 180 nm, 1.8-V CMOS. A linear scaling of throughput with feature size gives an information throughput of 444 Mbps and thus, an energy per decoded-bit of 2.5 nJ. Moreover, 52.5 mm^2 of silicon area is dedicated just for the decoder in [1]. We could potentially fit more than three of our decoders in the same area attaining decoded throughputs in excess of 500 Mbps.

Darabiha implemented a (2048,1723) RS-based (Reed-Solomon) Gallager (6,32)-regular LDPC-BC using a hard-decision MPA to reduce interconnect complexity [63]. This design operated at 3.2 Gbps of coded throughput and is a higher-rate code (rate = 0.841). This makes the comparison difficult as higher-rate codes and hard-decision decoders are more conducive to high throughputs. Such decoders require fewer computations per information bit. However, this decoder was a hard decoder operating on single-bit LLRs. Hence, error correcting performance was significantly degraded with respect to a soft-decision decoder.

Mansour's design is another LDPC-BC decoder but based on a Turbo decoding algorithm [64]. That design also used a 180 nm, 1.8-V CMOS process but the reported results are only from simulation. That LDPC-BC decoder targeted a 2304-b, rate-2/3, irregular code with 10 iterations per frame. It is assumed to have 128 Mbps of decoded throughput with an average power consumption of 1.2 W and hence, an energy per decoded-bit of 9.2 nJ. In addition, this design does not scale well with more iterations. This means that each additional iteration reduces the decoder's throughput, whereas only the initial latency increases in our LDPC-CC decoder design.

Several other digital LDPC-BC implementations exist and Table 6.4 summarizes the facts for a select list of more recent work on FPGA and ASIC platforms.

6.5 Summary

We described a suitable architecture employing LDPC-CCs and test results indicate proper operation at 175 MHz while achieving a BER of approximately 10^{-3} at 3 dB SNR. This does not compare well against existing work but it was noted that the allotted silicon area limited the placement of only 10 processors. The fabricated ASIC did not deviate from the straightforward solution and thus, there are several areas for improvement.

Chapter 7

Conclusion

The purpose of forward error control coding is to augment messages to produce codewords containing deliberately introduced redundancy, or code-bits. With proper care, these *bits* can be added in such a way that valid codewords are sufficiently distinct from each other. This is so the transmitted message can be correctly inferred at the receiver, even when some of the bits are corrupted during transmission over the channel.

Low-density parity-check (LDPC) codes do just that and they do it very well. After their independent rediscovery, they are now considered to be among the most powerful class of forward error correction (FEC) codes since Turbo codes. In fact, LDPC codes are capable of performance extraordinarily close to the Shannon limit when appropriately decoded.

The sum-product algorithm operating on the factor graph (bipartite graph) uses message-passing in the decoding of LDPC codes and furthermore, specific instances of the same algorithm can also operate on suitably defined factor graphs in the forward/backward algorithm, the Viterbi algorithm, the iterative decoding of Turbo codes and on parallel concatenated convolutional codes. With the flexibility of LDPC block codes (LDPC-BCs), codes can be constructed to exactly match a particular block size or code rate, though practical implementations may impose certain constraints on block sizes and/or obtainable code rates.

LDPC convolutional codes (LDPC-CCs) are a recently introduced variant that uphold the same properties as its block-oriented counterpart. They offer an alternative implementation targeting applications with streaming data service or continuous input/output requirements, i.e. it can handle variable-length blocks. The simple, iterative decoder architecture of LDPC-CCs simplifies design scalability and avoids severe global routing issues, which is a major problem with LDPC-BCs decoders. Unfortunately, they have not been studied as well as LDPC-BCs and thus, warrant further research in the area. With that notion, the popularity of Gallager's decoding algorithm extends far beyond information theorists. The presumption of an iterative algorithm operating on a graph has been generalized and is now capable of unifying a wide range of different algorithms from the domains of digital communications

and signal processing, among several others. This in itself helps evolve the field of FEC and LDPC coding.

7.1 Summary of Results

Chapter 1 begins the thesis with the motivation behind FEC while Chapters 2 and 3 express the mathematical background of LDPC-BCs and LDPC-CCs, respectively. In Chapter 4, we summarize some prior work in the area of digital and analog decoding. The main focus of this thesis resides in Chapters 5 and 6, where we discuss an application-specific integrated circuit (ASIC) implementation and the test results reported by the physical chip. Here, we describe the first known ASIC implementation of a LDPC-CC encoder and decoder.

The chip implements a rate-1/2, (128,3,6) code and provides a decoded information throughput of 175 Mbps. It dissipates 1.3 W of power, from a 1.8-V supply, in a low noise environment. The LDPC-CC decoder employs a soft-decision, message-passing algorithm that executes on a unidirectional cascade of ten identical processing units. The ASIC was not especially power-efficient because of the register-intensive architecture that was adopted for design convenience. Significantly reduced power and increased decoding throughput should be achievable using customized multi-port memories that are matched at the layout level with the associated processor logic.

Other areas for improvement include a detailed debug interface. The fabricated version did not encompass several test modes and therefore, it was not possible to pinpoint the exact location of any errors, if and when any occurred.

7.2 Feasibility

Fig. 7.1 exists to illustrate that a (2048,3,6) LDPC-CC is comparable in performance to the (100000,3,6) LDPC-BC. However, it should be noted that the $N = 100,000$ is practically infeasible to implement, given the current technology. Nonetheless, the implementation of the (2048,3,6) was possible and one architecture is stated in [68]. In that article, the *termination* overhead circuitry is also included and hence, is tuned for today's practical, packet-based applications. Moreover, recent work has shown that it is possible to reduce the size of the overhead [44].

7.3 Potential Applications

It has been shown that LDPC-CCs may be better suited to certain applications, such as streaming video and variable-length, packet-switching networks, than LDPC-BCs [55]. Another advantage is that the packet encoding and decoding can be

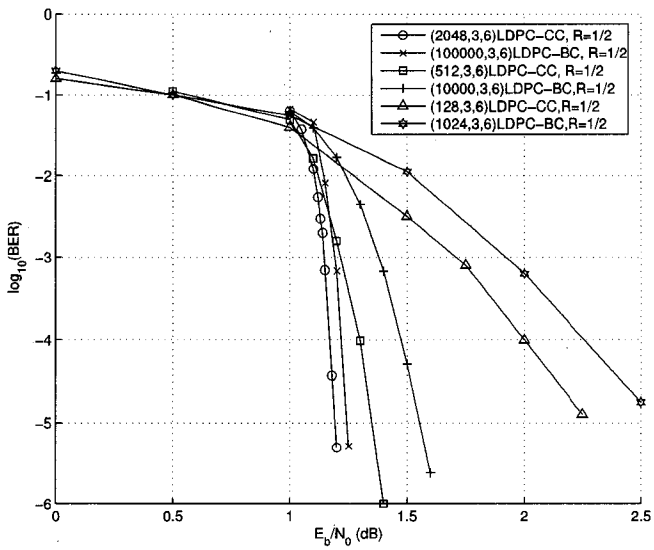


Figure 7.1: Plot showing a BER comparison of a (2048,3,6) versus a (100000,3,6) LDPC-BC, among several others. *[Graph courtesy of Zhengang Chen, University of Alberta.]*

done starting from a known state. This constraint can be utilized at the decoder to greatly reduce the probability of a bit error over the first few hundred bits of the packet. This is advantageous because in most packet formats, important routing information is located in the beginning or header of the packet. An example of this is the destination and source media access control (MAC) addresses in the Ethernet frame. Providing enhanced error protection over these important fields is advantageous for low-latency cut-through switching [69]. In this type of switching, packets are routed as soon as their destination address is evaluated and before they are checked for errors. As the address is more likely to be error free, the dangers associated with cut-through switching are minimized.

For storage applications such as compact discs, digital versatile discs and hard-drives, most use Reed-Solomon codes for error-correction. Their popularity is due to their robustness against bursts of errors and the availability of efficient decoding algorithms. However, future storage applications require dramatic increases in storage capacity and consequently higher data transfer rates. To achieve the desired performance with Reed-Solomon codes, it would require a more complex decoder and as storage capacities increase continually, it will eventually become infeasible to use these codes. LDPC codes could be the alternative coding scheme since they have excellent error correcting capabilities at high code rates and decoding complexity that is linear with frame size.

7.4 Future work

There are several topics to explore including code construction techniques, characterization and performance comparisons with respect to other channel types (erasure, fading, etc.) and VLSI research in search of architectural improvements.

It is known that some of the most powerful codes with low error floors are irregular codes. However, randomly constructed irregular codes often experience drastic implementation issues since the bipartite node degrees vary greatly. A regular structure with patterned connections help with the implementation but at the expense of error performance. Therefore, more work is needed to find an algebraic code construction technique to assist hardware designers and still achieve the randomly constructed code's error performance.

Most of the literature to-date has concentrated on an additive white Gaussian noise (AWGN) channel with binary phase-shift keying (BPSK) modulation. Here, the transmitted message is always seen at the receiver but probably corrupted by noise in the channel. It is not well understood how LDPC codes perform in an environment where erasures and packet loss are unavoidable. Therefore, further research is required to map out the error correcting capability of LDPC codes under different channels, including using different modulation/demodulation techniques.

The VLSI research portion can potentially foresee huge advantages, especially for industry. The parity-check and variable nodes are replicated countless times so it becomes worthwhile to optimize the node's function. This can include using the tanh-based parity operation to obtain the added error performance instead of the min-sum approach. In addition, optimization of basic components such as adder circuits become key. Therefore, the inevitable search for lower power, smaller area and increased speed architectures are a must.

As an aside, Appendix C lists a tutorial that explores LDPC-CCs using complementary ferroelectric-capacitor logic. It is based around the research of [70, 71] incorporating content-addressable memory modules in the decoding process. The entire tutorial and all text/figures/ideas given within was kindly provided by Dr. Chris Winstead of Utah State University. For more information, see [72].

7.5 Final Remarks

The capacity-approaching coding performance of LDPC-CCs is comparable to LDPC-BCs, Turbo codes and Turbo Product codes. LDPC-BCs are being considered as the FEC code of choice in next-generation communication standards such as 802.16e (WiMAX), 802.11n (Wi-LAN) and 802.3an (10GBASE-T). It has been adopted for the digital video broadcasting satellite (DVB-S2) standard. As LDPC-BC implementations become more widespread, the perceived future for LDPC-CCs will likely take its course. It is just a matter of time!

Bibliography

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

[2] Altera Corporation, "Accelerating WiMAX system design with FPGAs," in *Altera Corporation White Paper*. Altera Corporation, October 2004, Version 1.0; Reference: WP-FPGA102204-1.0.

[3] A. J. Felström and K. Sh. Zigangirov, "Time-varying periodic convolutional codes with low-density parity-check matrix," *IEEE Trans. on Information Theory*, vol. 45, no. 6, September 1999.

[4] H.-A. Loeliger, F. Lustenberger, M. Helfenstein, and F. Tarkoy, "Probability propagation and decoding in analog VLSI," *IEEE Trans. on Information Theory*, vol. 47, no. 2, pp. 837–843, February 2001.

[5] Chris Winstead, Nhan Nguyen, Vincent C. Gaudet, and Christian Schlegel, "Low-voltage CMOS circuits for analog iterative decoders," *IEEE Trans. on Circuits and Systems I (TCASI)*, vol. 53, no. 4, pp. 829–841, April 2006.

[6] S. Bates and G. Block, "A memory-based architecture for FPGA implementations of low-density parity-check decoders," in *Proc. of IEEE Symp. on Circuits and Systems (ISCAS'05)*, May 2005, vol. 1.

[7] W. Cary Huffman and Vera Pless, *Fundamentals of Error Correcting Codes*, Cambridge University Press, 2003, Online book: Books24x7.

[8] Roger Freeman, "Noise," 2005, This is an electronic document available from `http://searchnetworking.techtarget.com/generic/0,295582,sid7_gci1071926%,00.html?bucket=ETA&topic=299303`. Date of publication: March 28, 2005. Date retrieved: December 25, 2006. Date last

modified: [unavailable].

[9] Werner Feibel, *The Network Press Encyclopedia of Networking*, Sybex, third edition, 2000, Online book: Books24x7.

[10] B. P. Lathi, *Modern Digital and Analog Communication Systems*, Oxford University Press, third edition, 1998.

[11] Charan Langton, "Intuitive guide to principles of communications," This is an electronic tutorial available from `www.complextoreal.com`. Date of publication: [unavailable]. Date retrieved: December 24, 2006. Date last modified: [unavailable]., 2006.

[12] Yugeng Zhao and M. S. Hsiao, "Reducing power consumption by utilizing retransmission in short range wireless network," in *Proc. of the 27th Annual IEEE Conference on Local Computer Networks (LCN 2002)*, November 2002.

[13] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, July and October 1948.

[14] Don Johnson, "Structure of communication systems," 2000, This is an electronic document available from `http://cnx.org/content/m0002/latest/#commsys`. Date of publication: June 20, 2000. Date retrieved: December 23, 2006. Date last modified: August 17, 2004.

[15] Wikipedia, "Noisy channel coding theorem — wikipedia, the free encyclopedia," 2006, This is an electronic document available from `http://en.wikipedia.org/w/index.php?title=Noisy_channel_coding_theorem&%oldid=94131721`. Date of publication: [unavailable]. Date retrieved: December 24, 2006. Date last modified: December 13, 2006.

[16] T.A. Welch, "A technique for high-performance data compression," *Proc. of the IEEE Computer Society*, vol. 17, no. 6, pp. 8–19, June 1984.

[17] Gregory K. Wallace, "The JPEG still picture compression standard," *Proc. of the IEEE Trans. on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, February 1992.

[18] L. H. Charles Lee, *Error-Control Block Codes for Communication Engineers*,

Artech House, 2000, Online book: Books24x7.

[19] IEEE Computer Society and IEEE Microwave Theory & Techniques Society, "IEEE standard for local and metropolitan area networks part 16: Air interface for fixed and mobile broadband wireless access systems amendment 2: Physical and medium access control layers for combined fixed and mobile operation in licensed bands and corrigendum 1," *Proc. of the IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005 (Amendment and Corrigendum to IEEE Std 802.16-2004)*, pp. 1–822, February 2006.

[20] Jungnam Yun and Mohsen Kavehrad, "PHY/MAC cross-layer issues in mobile WiMAX," *Bechtel Telecommunications Technical Journal*, vol. 4, no. 1, pp. 45–56, January 2006.

[21] B. P. Lathi, *Signal Processing & Linear Systems*, Berkeley-Cambridge Press, 1998.

[22] Warren Hioki, *Telecommunications*, Prentice Hall, fourth edition, 2001.

[23] Shu Lin and Daniel J. Costello Jr., *Error Control Coding*, Fundamentals and Applications. Pearson Prentice Hall, second edition, 2004.

[24] Wayne Tomasi, *Electronic Communications Systems: Fundamentals through Advanced*, Prentice Hall, fourth edition, 2001.

[25] Richard W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. XXVI, no. 2, pp. 147–160, April 1950.

[26] Dmitri Truhachev, *On the Construction and Analysis of Iteratively Decodable Codes*, Ph.D. thesis, Lund University, Lund, Sweden, 2004.

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

[28] Sae-Young Chung, G. D. Forney Jr., T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Communication Letters*, vol. 5, no. 2, pp. 58–60, February 2001.

[29] R. Michael Tanner, "A recursive approach to low complexity codes," *Proc. of the IEEE Trans. on Information Theory*, vol. 27, no. 5, pp. 533–547, Septem-

ber 1981.

[30] Bernhard M. J. Leiner, "LDPC codes – a brief tutorial," This is an electronic tutorial available from `http://users.tkk.fi/~pat/coding/essays/ldpc.pdf`. Date of publication: April 8, 2005. Date retrieved: January 7, 2007. Date last modified: [unavailable]., 2005.

[31] David J. C. Mackay, Simon T. Wilson, and Matthew C. Davey, "Comparison of constructions of irregular gallager codes," *IEEE Trans. on Communications*, vol. 47, no. 10, pp. 1449–1454, October 1999.

[32] Thomas J. Richardson, M. Amin Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. on Information Theory*, vol. 47, no. 2, pp. 619–637, February 2001.

[33] T. J. Richardson and R. L. Urbanke, "Efficient Encoding of Low-Density Parity-Check Codes," *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 638–656, February 2001.

[34] David J. C. Mackay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. on Information Theory*, vol. 45, no. 2, pp. 399–431, March 1999.

[35] The Institute of Electrical and Electronic Engineers, *IEEE Standard 802.11-2006 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*, The IEEE, 2006.

[36] The Institute of Electrical and Electronic Engineers, *IEEE Standard 802.3-2002 : Carrier Sense Multiple Access with Collision Detection CSMA/CD Access Methods and Physical Layer Specification*, The IEEE, 2005.

[37] P. Elias, "Coding for noisy channels," *IRE Convention Record*, vol. 4, pp. 37–47, March 1955.

[38] C. Schlegel and L. Perez, *Trellis and Turbo Coding*, IEEE, first edition, 2004.

[39] Daniel J. Costello Jr., Ali Emre Pusane, Stephen Bates, and Kamil Sh. Zigangirov, "A comparison between LDPC block and convolutional codes," in *Proc. of IEEE Information Theory and Applications Workshop*, San Diego, CA, USA, February 2006.

[40] Zhengang Chen and Stephen Bates, "Construction of low-density parity-check convolutional codes through progressive edge-growth," *IEEE Communication Letters*, vol. 9, no. 12, pp. 1058–1060, December 2005.

[41] R. Tanner, D. Sridhara, A. Sridharan, T. Fuja, and D. Costello Jr., "LDPC block and convolutional codes based on circulant matrices," *IEEE Trans. on Information Theory*, vol. 50, no. 12, pp. 2966–2984, December 2004.

[42] A. Sridharan and D. Costello Jr., "A new construction method for low density parity check convolutional codes," in *Proc. of the IEEE Information Theory Workshop*, October 2002.

[43] Arvind Sridharan, *Design and Analysis of LDPC Convolutional Codes*, Ph.D. thesis, University of Notre Dame, February 2005.

[44] Zhengang Chen, Tyler L. Brandon, Stephen Bates, Duncan G. Elliott, and Bruce F. Cockburn, "Efficient implementation of low-density parity-check convolutional code encoders with termination," *to be submitted to IEEE Transactions on Circuits and Systems I (TCASI)*, February 2007.

[45] Stephen Bates, Duncan G. Elliott, and Ramkrishna Swamy, "Termination sequence generation circuits for low-density parity-check convolutional codes," *IEEE Transactions on Circuits and Systems I (TCASI)*, vol. 53, no. 9, pp. 1909–1917, September 2006, See also IEEE Transactions on Fundamental Theory and Applications.

[46] Zhengang Chen, Stephen Bates, and Xiaodai Dong, "Low-density parity-check convolutional codes applied to packet based communication systems," in *Proc. of IEEE Global Telecommunications Conference (GLOBECOM'05)*, December 2005, vol. 3, pp. 1250–1254.

[47] N. Wiberg, *Codes and Decoding on General Graphs*, Ph.D. thesis, Linkoping University, Linkoping, Sweden, 1996.

[48] T. Zhang and K. Parhi, "A 54 MBPS (3,6)-regular FPGA LDPC decoder," in *Proc. of the IEEE Workshop on Signal Processing*, October 2002.

[49] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low-density parity-check codes," *IEE Electronic Letters*, vol. 32, no. 18, pp. 1645–

1646, August 1996.

[50] Andrew Schaefer, Matthias Moerz, Arvind Sridharan, and Daniel J. Costello Jr., "Analog rotating ring decoder for an ldpc convolutional code," in *Proc. of IEEE Information Theory Workshop*, April 2003, pp. 226–229.

[51] Chris Winstead and Christian Schlegel, "Analog decoding: the state of the art," in *Proc. of IEEE Eighth International Symp. on Spread Spectrum Techniques and Applications (ISSSTA)*, September 2004, pp. 503–511.

[52] Ramkrishna Swamy, Stephen Bates, and Tyler Brandon, "Architectures for ASIC implementations of low-density parity-check encoders and decoders," in *Proc. of IEEE Symp. on Circuits and Systems (ISCAS'05)*, May 2005, vol. 5.

[53] Ramkrishna Swamy, Stephen Bates, Tyler Brandon, Bruce F. Cockburn, Duncan G. Elliott, John Koob, and Zhengang Chen, "Design and test of a rate-1/2 (128,3,6) low-density parity-check convolutional code encoder and decoder," *accepted for publication in the IEEE Journal of Solid-State Circuits (JSSC)*, 2007.

[54] D. E. Knuth, *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*, Addison-Wesley, third edition, 1997.

[55] S. Bates, Z. Chen, and X. Dong, "Low-density parity check convolutional codes for packet switching networks," *Submitted to IEEE Communication Letters*, 2004.

[56] Tyler L. Brandon, Bruce F. Cockburn, and Duncan G. Elliott, "HDL2GDS: A fully automated ASIC digital design flow," in *Proc. of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, May 2005, pp. 1535–1538.

[57] Lund University, *Digital ASIC Design: A Tutorial on the Design Flow*, October 2005.

[58] Tyler Brandon, Robert Hang, Gary Block, Vincent Gaudet, Bruce F. Cockburn, Sheryl Howard, Christian Giasson, Keith Boyle, Paul Goud, S. Sheikh Zeinoddin, Anthony Rapley, Stephen Bates, Duncan G. Elliott, and Christian Schlegel, "A 250-Mbps Min-Sum LDPC Decoder Using Bit-Serial Message

Exchange," *in preparation*, 2007.

[59] Joseph Di Giacomo, *Designing with High Performance ASICs*, Prentice Hall, Inc., 1992.

[60] Tyler L. Brandon *et al.*, "ICFAALP3: LDPC-CC design in 90 nm CMOS technology," yet to be published, 2007.

[61] Altera Inc., *Stratix EP1S80 DSP Development Board Data Sheet, Ver. 1.3*, December 2004, Available at `http://altera.com/literature/ds/ds_stratix_dsp_bd_pro.pdf`.

[62] Brandon Waldo and Jim Flynn, "Techniques for power optimization," in *Compiler Issues*. Synopsys Inc., December 2004.

[63] Ahmad Darabiha, Anthony Chan Carusone, and Frank R. Kschischang, "Multi-gbit/sec low density parity check decoders with reduced interconnect complexity," in *Proc. of the IEEE International Symp. on Circuits and Systems (ISCAS05)*, May 2005, vol. 5.

[64] Mohammad M. Mansour and Naresh R. Shanbhag, "High-throughput LDPC decoders," *IEEE Trans. on Very Large Scale Integration Systems (VLSI)*, vol. 11, no. 6, pp. 976–996, December 2003.

[65] Zhiqiang Cui and Zhongfeng Wang, "A 170 Mbps (8176,7156) quasi-cyclic ldpc decoder implementation with FPGA," in *Proc. of the IEEE International Symp. on Circuits and Systems (ISCAS06)*, May 2006, pp. 5095–5098.

[66] Mohammad M. Mansour and Naresh R. Shanbhag, "A 640-Mb/s 2048-bit programmable LDPC decoder chip," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 41, no. 3, pp. 684–698, March 2006.

[67] Chien-Ching Lin, Kai-Li Lin, Hsie-Chia Chang, and Chen-Yi Lee, "A 3.33Gb/s (1200,720) low-density parity-check code decoder," in *Proc. of the 31st European Solid-State Circuits Conference (ESSCIRC)*, September 2005, pp. 211–214.

[68] Stephen Bates, Logan Gunthorpe, Ali Emre Pusane, Zhengang Chen, Kamil Zigangirov, and Daniel J. Costello Jr., "Decoders for low-density parity-check convolutional codes with large memory," in *Proc. of IEEE International Symp.*

on Circuits and Systems (ISCAS 2006), May 2006, pp. 5103–5106.

[69] R. Seifert, *The Switch Book*, John Wiley & Sons, Inc., first edition, 2000.

[70] Hiromitsu Kimura, Takahiro Hanyu, Michitaka Kameyama, Yoshikazu Fujimori, Takashi Nakamura, and Hidemi Takasu, "Complementary ferroelectric-capacitor logic for low-power logic-in-memory VLSI," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 6, pp. 919–926, June 2004.

[71] Ali Sheikholeslami, P. Glenn Gulak, and Takahiro Hanyu, "A multiple-valued ferroelectric content-addressable memory," in *Proc. of the IEEE 26th International Symposium on Multiple-Valued Logic*, May 1996, pp. 74–79.

[72] Chris Winstead, "Tutorial: Low-density parity-check convolutional codes," Available from `http://www.engineering.usu.edu/ece/faculty/winstead/`, December 2005.

Appendix A

Parity-Check Matrix: Code File & Parser

A.1 PCM File Parser

The following Python code (PCM-PythonCode.py) shows a method for parsing the parity-check matrix (PCM) file. To execute the code, run

```
python PCM_PythonCode.py
```

and it generates an output file, "llrfile.txt." This file contains all the LLRs required per phase and hence, for the (128,3,6) LDPC-CC, it shows seven reads and seven writes at each phase of the 129 phases.

Listing A.1: PCM-PythonCode.py

```
#! /usr/bin/python

################################################################################
# Author      : Ramkrishna Swamy
# Date        : September 12, 2005
#
# Description: This script analyzes a Parity-Check Matrix (PCM) file, which is
# used for storing the H matrix. It provides a parser to decode the PCM's
# information. In addition, it generates a "required" LLR list file. That is,
# for the (128,3,6) LDPC-CC, there will be 129 lines of output generated. Each
# line will have 14 LLRs: 7 reads and 7 writes. These are the LLRs that NEED
# to be modified and updated with the various functions/operations.
################################################################################

import pdb
import sys,copy,math,sets

# This simple parser takes only REGULAR codes!

config = {
    # the following configs MUST be set correctly to start
    'PCMfile':'./Ldpcc-128-1-2-3-6.pcm', # input PCM file
    'ReqLLRfile':'./llrfile.txt',        # output LLR file

    # the following will be automatically configured.
    # The current values are just DUMMIES, which will be overwritten
    'numRows': 258, # other words, rate-1/2 with 129 phases
    'codeRate':2,   # means rate-1/2
    'memory':128,   # m_s = 128
    'numOnesCol':6, # regular code with 6 ones per column
    'numOnesRow':3, # regular coded with 3 ones per row
    'numPhases':129 # # of phases = memory + 1 = m_s + 1
    }

# Calculates which rows the current phase is processing: variable nodes
```

```
def calculateAbsRow(rowOffset,phase):
    rownum = rowOffset - (config['codeRate']*phase)
    while rownum < 0:
        rownum += config['numRows']
    return rownum

# Reads the PCM file and stores all LLR information in Python dictionaries.
def readPCMfile():
    try:
        fh_pcm = open(config['PCMfile'], 'r')
        pcm_text = fh_pcm.readlines()
        fh_pcm.close
    except:
        print 'Could not open '+ config['PCMfile']
        print 'Exiting program...'
        sys.exit()

    line = pcm_text[0].split()
    config['memory']=int(line[0])
    config['codeRate']=int(line[2])
    config['numOnesRow']=int(line[3])
    config['numOnesCol']=int(line[4])

    line = pcm_text[1].split()
    config['numPhases']=int(line[0])
    config['numRows']=int(line[1])

    numOnesPhase = pcm_text[3].split()
    #offset = config['codeRate']
    phase = 0
    LLRbyPhase={}
    LLRbyRow={}


    for i in range(4,config['numPhases']*2+4,2) :
        positions = pcm_text[i].split()
        weights = pcm_text[i+1].split()
        #print positions, weights
        #print config
        LLRbyPhase[phase]=[]

        for j in range (0,int(numOnesPhase[phase])):
            #rownum = (offset-int(positions[j])) % config['numRows']
            rownum = calculateAbsRow(int(positions[j])-1, phase)
            wgt = int(weights[j])


            LLRvalue = (wgt * config['numRows']) + rownum

            if LLRbyRow.has_key(rownum):
                LLRbyRow[rownum].append(LLRvalue)
            else:
                LLRbyRow[rownum]=[rownum, LLRvalue]
            LLRbyPhase[phase].append(LLRvalue)

        #offset += config['codeRate']
        phase += 1

    print 'Finished reading and analyzing PCM file: ' + config['PCMfile']
    return (LLRbyRow, LLRbyPhase, numOnesPhase)

# Generates all the LLRs for reads and writes for every phase
def genReqLLRfile(LLRbyRow, LLRbyPhase, numOnesPhase):

    allLLRs = []
    perPhaseInfo = {}
    allVN_key = 'allVN'

    fh_out = open(config['ReqLLRfile'],'w')
    for ph in range(0,config['numPhases']):
        allLLRs.append([])
        perPhaseInfo[ph] = {}
        perPhaseInfo[ph]['reads'] = []
        perPhaseInfo[ph]['writes'] = []
        perPhaseInfo[ph]['2pcn'] = []
        perPhaseInfo[ph]['pcn2vn'] = []
        perPhaseInfo[ph]['pcn'] = []
        perPhaseInfo[ph][allVN_key] = []

        for pcn_pos in range(0,int(numOnesPhase[ph])):
            allLLRs[ph].append(LLRbyPhase[ph][pcn_pos])
            perPhaseInfo[ph]['pcn'].append(LLRbyPhase[ph][pcn_pos])

        vn_toprow = calculateAbsRow(-config['codeRate'],ph)
        # Instead of the above, the following work too!
        #vn_toprow = ((ph+1)*config['codeRate']) % config['numRows']

        for vn_row in range(vn_toprow,vn_toprow+config['codeRate']):
            vn_key = 'vn'+str(vn_row-vn_toprow)
```

```
            perPhaseInfo[ph][vn_key] = []

            for vn_pos in range(0,config['numOnesRow']+1):
                allLLRs[ph].append(LLRbyRow[vn_row][vn_pos])
                perPhaseInfo[ph][vn_key].append(LLRbyRow[vn_row][vn_pos])
                perPhaseInfo[ph][allVN_key].append(LLRbyRow[vn_row][vn_pos])

        perPhaseInfo[ph]['reads']=copy.deepcopy(allLLRs[ph])
        fh_out.writelines(" ".join(["%s" % (k) for k in allLLRs[ph]]))
        fh_out.write("\n")

    fh_out.close
    print 'Finished generating initial REQUIRED LLR list: ' + config['ReqLLRfile']
    return (allLLRs, perPhaseInfo)

#### MAIN PROGRAM ####
# We first read the Parity-Check Matrix (PCM) file;
# This creates the resulting Python dictionaries
(LLRbyRow,LLRbyPhase,numOnesPhase) = readPCMfile()
# We then generate an OUTPUT file that shows the LLRs and which of them
# (reads/writes) are referenced per phase.
(allLLRs, perPhaseInfo) = genReqLLRfile(LLRbyRow,LLRbyPhase,numOnesPhase)
```

A.2 (128,3,6) rate-1/2, LDPC-CC

Listing A.2: Ldpcc-128-1-2-3-6.pcm

```
128 1 2 3 6
129 258
6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 →
    ↪6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 →
    ↪6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
1 257 212 132 23 18
1 3 3 2 2 1
1 257 244 108 66 19
1 3 3 2 2 2
1 257 218 126 82 59
1 3 3 2 2 2
1 257 208 148 41 26
1 3 3 3 2 1
1 257 228 116 77 52
1 3 3 2 2 2
1 257 232 156 57 20
1 3 3 3 2 1
1 257 252 122 78 45
1 3 3 2 2 2
6 1 257 142 63 62
1 1 3 2 2 2
1 257 192 160 69 46
1 3 3 3 2 1
1 257 254 112 79 50
1 3 3 2 2 1
1 257 252 150 106 85
1 3 3 3 2 2
24 1 257 180 106 31
1 1 3 2 2 2
1 257 234 138 51 40
1 3 3 2 2 1
16 1 257 164 68 45
1 1 3 2 2 2
1 257 224 178 84 31
1 3 3 3 2 2
1 257 254 192 103 90
1 3 3 3 2 2
1 257 240 200 72 63
1 3 3 3 2 2
1 257 220 158 72 71
1 3 3 2 1 2
1 257 252 196 88 43
1 3 3 3 1 2
1 257 222 134 117 44
1 3 3 2 2 1
16 1 257 158 79 52
1 1 3 3 2 1
2 1 257 182 104 87
1 1 3 3 2 2
1 257 242 156 129 58
1 3 3 3 2 1
1 257 252 202 100 89
1 3 3 3 2 2
```

1 257 242 214 82 73
1 3 3 3 1 2
48 1 257 176 138 107
1 1 3 3 2 2
1 257 252 188 127 102
1 3 3 2 2 2
1 257 232 174 126 125
1 3 3 2 1 2
26 1 257 190 137 80
1 1 3 2 2 1
30 1 257 156 99 86
1 1 3 2 2 1
1 257 248 212 127 68
1 3 3 3 2 1
50 1 257 178 145 136
1 1 3 2 2 2
60 1 257 236 140 123
1 1 3 3 2 2
52 1 257 230 153 70
1 1 3 3 2 1
1 257 248 242 104 89
1 3 3 3 1 2
64 1 257 162 138 121
1 1 3 2 1 2
18 1 257 220 154 87
1 1 3 3 2 2
42 1 257 174 151 96
1 1 3 2 2 1
8 1 257 178 156 89
1 1 3 2 2 2
2 1 257 248 124 107
1 1 3 3 1 2
14 1 257 184 150 143
1 1 3 2 2 2
62 1 257 172 108 93
1 1 3 2 2 2
84 1 257 238 142 89
1 1 3 2 2 2
56 55 1 257 224 96
2 2 1 3 3 2
75 6 1 257 246 92
2 1 1 3 3 2
68 23 1 257 186 142
1 2 1 3 3 2
94 23 6 1 257 162
2 2 1 1 3 3
83 22 1 257 198 116
2 1 1 3 3 2
82 31 1 257 198 130
2 2 1 3 3 2
65 60 1 257 230 110
2 1 1 3 3 2
50 29 1 257 254 118
1 2 1 3 3 2
97 36 1 257 222 116
2 2 1 3 3 2
95 62 1 257 202 170
2 1 1 3 3 3
107 62 1 257 234 138
2 1 1 3 3 2
67 40 1 257 244 164
2 2 1 3 3 3
57 30 14 1 257 146
2 1 1 1 3 2
72 69 22 1 257 166
2 2 1 1 3 3
107 38 1 257 236 188
2 2 1 3 3 3
67 62 1 257 230 196
2 2 1 3 3 3
71 48 1 257 252 178
2 1 1 3 3 3
103 86 4 1 257 208
2 1 1 1 3 3
118 99 1 257 216 180
2 2 1 3 3 3
64 41 1 257 248 152
1 2 1 3 3 2
99 62 1 257 226 156
2 1 1 3 3 2
120 89 16 1 257 200
2 2 1 1 3 2
78 73 36 1 257 198
1 2 1 1 3 2
122 117 1 257 238 214
2 2 1 3 3 3
132 83 1 257 250 180
2 2 1 3 3 2
120 81 38 1 257 220

1 2 1 1 3 3
109 102 32 1 257 202
2 1 1 1 3 3
62 61 1 257 232 146
1 2 1 3 3 2
86 69 1 257 250 158
1 2 1 3 3 2
143 86 1 257 234 220
2 1 1 3 3 3
134 85 1 257 256 208
2 2 1 3 3 3
124 85 34 1 257 174
2 2 1 1 3 3
147 88 10 1 257 170
2 1 1 1 3 2
104 71 50 1 257 200
1 2 1 1 3 3
148 117 8 1 257 162
2 2 1 1 3 2
110 97 46 1 257 234
1 2 1 1 3 3
139 138 70 1 257 202
2 2 1 1 3 3
132 125 76 1 257 198
2 2 1 1 3 2
141 136 44 1 257 204
2 1 1 1 3 3
132 89 56 1 257 214
2 2 1 1 3 3
121 92 34 1 257 190
2 1 1 1 3 2
150 91 40 1 257 208
1 2 1 1 3 3
170 145 78 1 257 256
2 2 1 1 3 3
128 84 37 1 257 240
2 2 2 1 3 3
134 82 47 1 257 196
3 2 2 1 3 3
142 67 18 1 257 228
2 2 1 1 3 3
158 44 19 1 257 250
3 1 2 1 3 3
126 87 38 1 257 216
3 2 1 1 3 3
168 81 54 1 257 194
3 2 2 1 3 3
124 27 18 1 257 188
2 2 1 1 3 3
176 87 54 1 257 220
3 2 2 1 3 3
162 64 63 12 1 257
2 1 2 1 1 3
178 27 22 1 257 208
3 2 1 1 3 3
184 42 23 1 257 222
3 1 2 1 3 3
172 89 88 1 257 208
2 2 2 1 3 3
138 93 52 1 257 242
2 2 1 1 3 3
132 83 72 1 257 214
3 2 1 1 3 3
144 100 59 4 1 257
2 1 2 1 1 3
164 63 56 1 257 240
2 2 2 1 3 3
188 113 64 34 1 257
2 2 2 1 1 3
128 90 51 1 257 212
2 2 2 1 3 3
144 123 114 1 257 240
2 2 2 1 3 3
186 106 61 1 257 234
3 1 2 1 3 3
166 99 50 28 1 257
2 2 1 1 1 3
210 102 67 22 1 257
3 2 2 1 1 3
188 118 71 12 1 257
3 2 2 1 1 3
182 128 101 18 1 257
2 2 2 1 1 3
214 136 87 1 257 248
3 2 2 1 3 3
180 71 70 24 1 257
2 2 1 1 1 3
226 104 103 8 1 257
3 1 2 1 1 3

```
208  89  62  44   1  257
2   2   1   1   1   3
160 132 105   1 257 236
3   2   2   1   3   3
156 111  92  12   1 257
2   2   1   1   1   3
180 122 121   1 257 252
2   2   2   1   3   3
204 126 105  20   1 257
3   2   2   1   1   3
160 114  93  26   1 257
3   1   2   1   1   3
168 152 149  36   1 257
2   2   2   1   1   3
168 122 109  46   1 257
2   2   2   1   1   3
242 112  75   1 257 252
3   1   2   1   3   3
164 157  84  64   1 257
2   2   1   1   1   3
244 110  85  14   1 257
3   1   2   1   1   3
216 134  95  74   1 257
3   2   2   1   1   3
200 143  96  78   1 257
2   2   1   1   1   3
190  96  87  64   1 257
2   1   2   1   1   3
172 157 106  46   1 257
2   2   1   1   1   3
257 208 161 154  78   1
3   2   2   2   1   1
```

Appendix B

ICFAALP2: VHDL Source Code

Project File List:

Listing B.1: LdpccPackageForIcfaalp2.vhd

```
--------------------------------------------------------------------------------
--
--    Filename: LdpccPackageForLcfaalp2.vhd
--
--    Created:  29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------------
--
--   Description
--   -----------
--
--  This is the top-level package file for the ICFAALP2 device which is to be
--  fabricated by TSMC 0.18u. It should replace the ususal package file and is
--  located in the ICFAALP2 directory along with the top level file.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

package LdpccPackageForIcfaalp2 is

   -- Declare the width of the inputs from pins here.

constant NUM_HIGH_SPEED_IPS : integer := 20; -- Number of high speed input pins.
constant NUM_HIGH_SPEED_OPS : integer := 16; -- Number of high speed output pins.
constant NUM_CONTROL_BITS   : integer := 18; -- Width of control bus.
                                             --
                                             --
  -- Declare the noise control values here.

constant NOISE_OFF          : std_logic_vector := "000";
constant NOISE_ONE          : std_logic_vector := "001";
constant NOISE_TWO          : std_logic_vector := "010";
constant NOISE_THREE        : std_logic_vector := "011";
constant NOISE_FOUR         : std_logic_vector := "100";
constant NOISE_FIVE         : std_logic_vector := "101";
constant NOISE_SIX          : std_logic_vector := "110";
constant NOISE_SEVEN        : std_logic_vector := "111";

  -- Declare the MSB and LSB positions of the LLRs here.

constant nLlrMagMsb : integer := 1;  -- The MSB of the absolute of the LoglikeRatios
constant nLlrMagLsb : integer := -2; -- The LSB of above
constant nLlrWidth  : integer := nLlrMagMsb-nLlrMagLsb+1;

  -- Declare some constants to do with the encoder and decoder here.

constant nCodeMemory          : integer := 129; -- The memory of the code
constant nCeilLog2CodeMemory : integer := 8;   -- The int larger than log2 of memory of the code
constant nNumParityChecks     : integer := 3;   -- No. of PCs per term
constant nProcessorLength     : integer := 140;  -- Length of delay line in processsor
constant nNumProcessors       : integer := 10;   -- Number of processors in decoder

-- WARNING: If you change this you must change the delay on phPhaseIn in the
-- processor unit!!!
constant nParityCheckLatency : integer  := 4;    -- Delay of parity check operation.

    -- Declare our types here. These make it easy to change our
    -- circuits from a central location.

subtype EncoderMemory is std_logic_vector ( nCodeMemory-1          downto 0 );
subtype Phase         is std_logic_vector ( nCeilLog2CodeMemory-1 downto 0 );
subtype LogicBit      is std_logic;

  -- Declare reset value for Phase. Also state the largest phase
  -- allowable which is usually equal to nCodeMemory-1.

constant PHASE_RESET   : Phase := (others=>'0');
constant PHASE_MAXIMUM : Phase := "10000000"; -- Base this on code memory

  -- Each decoder must have an phase offset associated with it that we
  -- load in at reset. This tells each processor how much to adjust
  -- their inputs by. The adjustment is
  --
  -- phase to use = phase in - offset mod PHASE_MAXIMUM
  --
  -- The offset for the 2nd processor should be nProcessorLength-nCodeMemory+
  -- nLlrSumLatency.

type PhaseArray is array ( nNumProcessors-1 downto 0)
```

```
      of Phase;
constant PHASE_ARRAY_RESET : PhaseArray := ( 0 =>"00000000" ,
                                             1 =>"00001101" ,--was 1110
                                             2 =>"00011010" ,
                                             3 =>"00100111" ,
                                             4 =>"00110100" ,
                                             5 =>"01000001" ,
                                             6 =>"01001110" ,
                                             7 =>"01011011" ,
                                             8 =>"01101000" ,
                                             9 =>"01110101" );
constant MaxProcNum : natural := 10;

  -- A type for the LLRs which contains storage space for the saturation
  -- bit (plus infinity), a sign bit and some magnitude bits.

type LogLikeRatio is record
      Sat : LogicBit;
      Sgn : LogicBit;
      Mag : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 );
end record LogLikeRatio;

  -- A subtype for the Delay Unit of the LLR delay line in the decoder.
  -- The width of this is the number of parity checks each term is involved in
  -- plus one.

type LogLikeRatioDelayUnit is array ( nNumParityChecks downto 0 )
        of LogLikeRatio;
type LogLikeRatioDelayUnitArray is array ( nNumProcessors-1 downto 0 )
    of LogLikeRatioDelayUnit;
type LogLikeRatioDelayLine is array ( nProcessorLength-1 downto 0 )
        of LogLikeRatioDelayUnit;

  -- Define the reset conditions for the LLRs. We build these up from the base unit
  -- and assign the sat bit to '1'.

constant LOG_LIKE_RATIO_RESET              : LogLikeRatio
        := (Sat=> '0' , Sgn => '0' , Mag => "0000");
constant LOG_LIKE_RATIO_DELAY_UNIT_RESET : LogLikeRatioDelayUnit
        := (others=>LOG_LIKE_RATIO_RESET);
constant LOG_LIKE_RATIO_DELAY_UNIT_ARRAY_RESET : LogLikeRatioDelayUnitArray
      := (others=>LOG_LIKE_RATIO_DELAY_UNIT_RESET);
constant LOG_LIKE_RATIO_DELAY_LINE_RESET : LogLikeRatioDelayLine
        := (others=>LOG_LIKE_RATIO_DELAY_UNIT_RESET);

  -- For the following declaration, this line didn't work in synthesis
  -- := (Sat=> '0' , Sgn => '1' , Mag => (others=>'0'));
  -- The work around was to hardcode the corect number of zeros in the Mag field.

constant LOG_LIKE_RATIO_NON_INF_VAL        : LogLikeRatio
        := (Sat=> '0' , Sgn => '1' , Mag => "0000");
constant LOG_LIKE_RATIO_DELAY_UNIT_NON_INF_VAL : LogLikeRatioDelayUnit
        := ( 0=>LOG_LIKE_RATIO_RESET, others=>LOG_LIKE_RATIO_NON_INF_VAL);

   -- In the partiy check node we want to associate input port id with the value
   -- so we create a new record.

type MinMaxTerm is record
      Llr : LogLikeRatio;
      Id  : std_logic_vector ( 2 downto 0 );
end record MinMaxTerm;
constant MIN_MAX_TERM_RESET : MinMaxTerm := ( Llr=>LOG_LIKE_RATIO_RESET , Id => "000" );

constant MAX_LLR_NOISE_WIDTH : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "1111";

  -- Declare some noise powers here.

constant LLR_MAX_MAG      : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "1111";
constant LLR_HALF_MAG     : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "1110";
constant LLR_10_32_MAG    : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "1100";
constant LLR_9_32_MAG     : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "1010";
constant LLR_QUARTER_MAG  : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "1000";
constant LLR_7_32_MAG     : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "0110";
constant LLR_3_16_MAG     : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "0100";
constant LLR_5_32_MAG     : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "0010";
constant LLR_EIGHTH_MAG   : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "0001";
constant LLR_ZERO_MAG     : std_logic_vector ( nLlrMagMsb-nLlrMagLsb downto 0 ) := "0000";

---------------------- NEWWWWWWWWWWWWWWWWW

constant PHASE_IN_DELAY : integer := 3;
subtype BlockCounter is std_logic_vector ( 41 downto 0 );
constant ERROR_WIN_MIN      : BlockCounter := "000000000000000000000000000000100000000000"; -- At least count →
    ↪ for 2048
constant ERROR_WIN_MAX      : BlockCounter := "100000000000000000000000000000000000000000"; -- At most count →
    ↪for a lot.
constant BLOCK_COUNTER_INC : BlockCounter := "000000000000000000000000000000000000000001";
```

```
constant NUM_ERRORS_WANTED_UPPER_THRESH : BlockCounter := "00000000000000000000000000010000000000"; -- →
    ↪ At most 1024 errors
constant NUM_ERRORS_WANTED_LOWER_THRESH : BlockCounter := "00000000000000000000000000001000000000"; -- →
    ↪ At least 512 errors

type NoiseArray is array (126 downto 0) of std_logic_vector (6 downto 0);

end LdpccPackageForIcfaalp2;
```

Listing B.2: Icfaalp2TopLevel.vhd

```
-------------------------------------------------------------------------------
--
--    Filename: Icfaalp1TopLevel.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ----------------------------------------------------------------------------
--
--   Description
--   -----------
--
-- This is the top-level VHDL file for the ICFAALP2 device which is to be
-- fabricated by TSMC 0.18u. It includes the followins subparts plus the
-- the top level pin-out which should map to a wrapper file. NOTE: There should
-- be no registers in this entity and all internal signals have the suffix Wire.
--
-- Note also that all signals coming from pins are suffixed Fp and those going to
-- pins are suffixed Tp. Note this is only pins at this level (e.g. a DLL might
-- be added). Also, we might need to adjust the clock signal naming based on
-- where it comes from.
--
-- The components are instantiated at this level.
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity Icfaalp2TopLevel is
    Port ( clkClockFp              : in std_logic;
           rstResetFp              : in std_logic;
           slvHighSpeedInputsFp    : in std_logic_vector ( NUM_HIGH_SPEED_IPS-1  downto 0 );
           slvControlFp            : in std_logic_vector ( NUM_CONTROL_BITS-1    downto 0 );
           --Outputs
           slvHighSpeedOutputsTp   : out std_logic_vector ( NUM_HIGH_SPEED_OPS-1 downto 0 );
           clkClockTp              : out std_logic
         );
end Icfaalp2TopLevel;

architecture Behavioral of Icfaalp2TopLevel is

-------------------------------------------------------------------------------
-- COMPONENTS
-------------------------------------------------------------------------------

    -- Components which we instantiate within the main
    -- block.

component InfoBitGenerator is
  generic (
    nNumBits: positive := 32
      );
  port (
    clkClock      : in  std_logic;          -- LFSR clock
    slLoad        : in  std_logic;          -- active high load
    rstReset      : in  std_logic;          -- active low reset
    slvSeed       : in  std_logic_vector(nNumBits-1 downto 0);  -- initial parallel seed value
    slvTestPattern: in  std_logic_vector(2 downto 0); -- type of test pattern to
                                                      -- realize
    phPhaseOut    : out Phase;
    slSerialOut : out std_logic);        -- serial output (rightmost register)
end component;

component LdpccEncoder is
    port (
       clkClock      : in LogicBit;
           rstReset      : in LogicBit;
```

```
            bitInfoBitIn   : in LogicBit;
            phPhaseIn      : in Phase;
            --Outputs
            bitInfoBitOut  : out LogicBit;
            bitCodeBitOut  : out LogicBit
        );
end component;

component LdpccProcessor is
    Port ( clkClock                 : in LogicBit;
           rstReset                 : in LogicBit;
           llrduInfoLlrDelayUnitIn  : in LogLikeRatioDelayUnit;
           llrduCodeLlrDelayUnitIn  : in LogLikeRatioDelayUnit;
           phPhaseIn                : in Phase;
           phPhaseOffset            : in Phase;
           --Outputs
           llrduInfoLlrDelayUnitOut : out LogLikeRatioDelayUnit;
           llrduCodeLlrDelayUnitOut : out LogLikeRatioDelayUnit
         );
end component;

component LdpccSignExtractor is
  Port ( clkClock              : in LogicBit;
         rstReset              : in LogicBit;
         llrduLlrDelayUnitIn   : in LogLikeRatioDelayUnit;
         --Outputs
         lbBitOut              : out LogicBit
           );
end component;

component ChannelEmulator is
    Port ( clkClock         : in LogicBit;
           rstReset         : in LogicBit;
           bitInfoBitIn     : in LogicBit;
           bitCodeBitIn     : in LogicBit;
           chemctrlControl  : in std_logic_vector ( 2 downto 0);
           -- Outputs
           llrInfoLlrOut    : out LogLikeRatio;
           llrCodeLlrOut    : out LogLikeRatio
         );
end component;

component InputInterface is
    Port ( -- Inputs from Pins
           clkClockFp              : in std_logic;
           rstResetFp              : in std_logic;
           slvHighSpeedInputsFp    : in std_logic_vector ( NUM_HIGH_SPEED_IPS-1  downto 0 );
           slvControlFp            : in std_logic_vector ( NUM_CONTROL_BITS-1    downto 0 );
           -- Common Outputs
           clkClock                : out LogicBit;
           rstReset                : out LogicBit;
           --Outputs to Info Bit Gen
           slLoadGenOut       : out std_logic;
           slvSeedGenOut      : out std_logic_vector ( 31 downto 0 );
           slvTestPtnGenOut   : out std_logic_vector (  2 downto 0 );
           -- Inputs from Info Bit Gen
           slvInfoBitGenIn    : in std_logic;
           phPhaseGenIn       : in Phase;
           -- Outputs to LdpccEncoder
           bitInfoBitOutEnc   : out LogicBit;
           phPhaseOutEnc      : out Phase;
           -- Inputs from LdpccEncoder
           bitInfoBitInEnc    : in LogicBit;
           bitCodeBitInEnc    : in LogicBit;
           -- Outputs to Channel Emulator
           bitInfoBitOutChE      : out LogicBit;
           bitCodeBitOutChE      : out LogicBit;
           ctrlcheControlChE     : out std_logic_vector ( 2 downto 0);
           -- Inputs from Channel Emulator
           llrInfoLlrInChE: in LogLikeRatio;
           llrCodeLlrInChE: in LogLikeRatio;
           -- Outputs to Output Interface
           llrduInfoLlrDelayUnitOutOpIf   : out LogLikeRatioDelayUnit;
           llrduCodeLlrDelayUnitOutOpIf   : out LogLikeRatioDelayUnit;
           phPhaseOutOpIf                 : out Phase;
           phaPhaseOffsetOutOpIf          : out PhaseArray
           );
end component;

component OutputInterface is
    Port ( -- Inputs from Pins
           slvControlFp            : in std_logic_vector ( NUM_CONTROL_BITS-1    downto 0 );
           -- Common Inputs
           clkClock                : in LogicBit;
           rstReset                : in LogicBit;
           -- Inputs from Encoder
           bitInfoBitInEnc    : in LogicBit;
           bitCodeBitInEnc    : in LogicBit;
           -- Inputs from Input Interface;
```

```
            llrduInfoLlrDelayUnitInIpIf   : in LogLikeRatioDelayUnit;
            llrduCodeLlrDelayUnitInIpIf   : in LogLikeRatioDelayUnit;
            phPhaseInIpIf                 : in Phase;
            phaPhaseOffsetInIpIf          : in PhaseArray;
            -- Inputs from Processors
            llrduaInfoLlrDelayUnitInPro   : in LogLikeRatioDelayUnitArray;
            llrduaCodeLlrDelayUnitInPro   : in LogLikeRatioDelayUnitArray;
            -- Inputs from Sign Extractor
            lbInfoBitInSgnEx              : in LogicBit;
            lbCodeBitInSgnEx              : in LogicBit;
            -- Inputs from Error Counter
            slvBlockCountLog2InErr        : in std_logic_vector (7 downto 0);
            -- Outputs to Processors
            phPhaseOutPro                 : out Phase;

            phaPhaseOffsetOutPro          : out PhaseArray;
            llrduaInfoLlrDelayUnitOutPro  : out LogLikeRatioDelayUnitArray;
            llrduaCodeLlrDelayUnitOutPro  : out LogLikeRatioDelayUnitArray;

            -- Outputs to Sign Extractors
            llrduInfoLleDelayUnitOutSignEx : out LogLikeRatioDelayUnit;
            llrduCodeLleDelayUnitOutSignEx : out LogLikeRatioDelayUnit;
            -- Outputs to Pins
            slvHighSpeedOutputsTp    : out std_logic_vector ( NUM_HIGH_SPEED_OPS-1 downto 0 )
            );
end component;

component ErrorCounter is
  port (
      -- inputs
      clkClock          : in  std_logic;          -- clock
      rstReset          : in  std_logic;          -- active high reset
      lbDecodedInfoBit  : in  LogicBit;
      -- outputs
      slvBlockCountLog2    : out std_logic_vector (7 downto 0));
end component;

-------------------------------------------------------------------------------
-- COMBINATORIAL
-------------------------------------------------------------------------------

    -- Combinatorial signals which are not assigned to registers. Since this is
    -- a top level block we suffix all these signals with the label Wire.

signal clkClockWire             : LogicBit;
signal rstResetWire             : LogicBit;
signal bitInfoBitInEncWire      : LogicBit;
signal phPhaseInEncWire         : Phase;
signal bitInfoBitOutEncWire     : LogicBit;
signal bitCodeBitOutEncWire     : LogicBit;
signal bitInfoBitInChEmWire     : LogicBit;
signal bitCodeBitInChEmWire     : LogicBit;
signal ctricheControlChEmWire   : std_logic_vector ( 2 downto 0);
signal llrInfoLlrOutChEmwire    : LogLikeRatio;
signal llrCodeLlrOutChEmWire    : LogLikeRatio;
signal llrduInfoLlrDelayUnitInOpIfWire    : LogLikeRatioDelayUnit;
signal llrduCodeLlrDelayUnitInOpIfWire    : LogLikeRatioDelayUnit;
signal phPhaseInOpIfWire                  : Phase;
signal llrduInfoLlrDelayUnitInSignExWire  : LogLikeRatioDelayUnit;
signal lbInfoBitOutSgnExWire              : LogicBit;
signal llrduCodeLlrDelayUnitInSignExWire  : LogLikeRatioDelayUnit;
signal lbCodeBitOutSgnExWire              : LogicBit;

  -- These next wires will have to added to and removed as we change the
  -- number of processors. Could done with arrays but safer at top level
  -- to do it this way.

signal phaPhaseOffsetInOpIfWire          : PhaseArray;
--signal phPhaseOffsetInOpIfWire          : Phase;
--signal phPhaseOffsetOutOpIfWire_1       : Phase;
--signal phPhaseOffsetOutOpIfWire_2       : Phase;
signal phaPhaseOffsetInProWire           : PhaseArray;
signal llrduaInfoLlrDelayUnitInProWire   : LogLikeRatioDelayUnitArray;
signal llrduaCodellrDelayUnitInProWire   : LogLikeRatioDelayUnitArray;
signal llrduaInfoLlrDelayUnitOutProWire  : LogLikeRatioDelayUnitArray;
signal llrduaCodeLlrDelayUnitOutProWire  : LogLikeRatioDelayUnitArray;
signal phPhaseInProWire                  : Phase;

signal slLoadGenInWire       : std_logic;
signal slvSeedGenInWire      : std_logic_vector ( 31 downto 0 );
signal slvTestPtnGenInWire   : std_logic_vector (  2 downto 0 );
signal slvInfoBitGenOutWire  : std_logic;
signal phPhaseGenOutWire     : Phase;
signal slvBlockCountLog2Wire    :  std_logic_vector (7 downto 0);

begin

-------------------------------------------------------------------------------
-- INSTANTIATIONS
```

```
InfoBitGenIcfaalp2 :  InfoBitGenerator
  generic map (  nNumBits=>32 )
  port map (
    clkClock        => clkClockWire,
    rstReset        => rstResetWire,
    slLoad          => slLoadGenInWire,
    slvSeed         => slvSeedGenInWire,
    slvTestPattern => slvTestPtnGenInWire,
    -- outputs
    slSerialOut     => slvInfoBitGenOutWire,
    phPhaseOut      => phPhaseGenOutWire
    );

InputInterfaceIcfaalp2 : InputInterface
    port map ( -- Inputs from Pins
           clkClockFp              => clkClockFp,
           rstResetFp              => rstResetFp,
           slvHighSpeedInputsFp    => slvHighSpeedInputsFp,
           slvControlFp            => slvControlFp,
           -- Common Outputs
           clkClock                => clkClockWire,
           rstReset                => rstResetWire,
           -- Outputs to LdpccEncoder
           bitInfoBitOutEnc        => bitInfoBitInEncWire,
           phPhaseOutEnc           => phPhaseInEncWire,
           --Outputs to Info Bit Gen
           slLoadGenOut       => slLoadGenInWire,
           slvSeedGenOut      => slvSeedGenInWire,
           slvTestPtnGenOut   => slvTestPtnGenInWire,
           -- Inputs from Info Bit Gen
           slvInfoBitGenIn    => slvInfoBitGenOutWire,
           phPhaseGenIn       => phPhaseGenOutWire,
           -- Inputs from LdpccEncoder
           bitInfoBitInEnc         => bitInfoBitOutEncWire,
           bitCodeBitInEnc         => bitCodeBitOutEncWire,
           -- Outputs to Channel Emulator
           bitInfoBitOutChE     => bitInfoBitInChEmWire,
           bitCodeBitOutChE     => bitCodeBitInChEmWire,
           ctrlcheControlChE    => ctrlcheControlChEmWire,
           -- Inputs from Channel Emulator
           llrInfoLlrInChE      => llrInfoLlrOutChEmWire,
           llrCodeLlrInChE      => llrCodeLlrOutChEmWire,
           -- Outputs to Output Interface
           llrduInfoLlrDelayUnitOutOpIf  => llrduInfoLlrDelayUnitInOpIfWire,
           llrduCodeLlrDelayUnitOutOpIf  => llrduCodeLlrDelayUnitInOpIfWire,
           phPhaseOutOpIf                => phPhaseInOpIfWire,
           phaPhaseOffsetOutOpIf         => phaPhaseOffsetInOpIfWire
           );

LdpccEncoderIcfaalp2 :  LdpccEncoder
   port map (
        clkClock       => clkClockWire,
        rstReset       => rstResetWire,
        bitInfoBitIn   => bitInfoBitInEncWire,
        phPhaseIn      => phPhaseInEncWire,
        bitInfoBitOut  => bitInfoBitOutEncWire,
        bitCodeBitOut  => bitCodeBitOutEncWire
               );

ChannelEmulatorIcfaalp2 : ChannelEmulator
    port map (
           clkClock       => clkClockWire,
           rstReset       => rstResetWire,
       bitInfoBitIn   => bitInfoBitInChEmWire,
           bitCodeBitIn   => bitCodeBitInChEmWire,
           chemctrlControl    => ctrlcheControlChEmWire,
           -- Outputs
           llrInfoLlrOut  => llrInfoLlrOutChEmWire,
           llrCodeLlrOut  => llrCodeLlrOutChEmWire
        );

LdpccInfoSignExtractorIcfaalp2 : LdpccSignExtractor
  port map (
        clkClock              => clkClockWire,
        rstReset              => rstResetWire,
     llrduLlrDelayUnitIn   => llrduInfoLlrDelayUnitInSignExWire,
        --Outputs
        lbBitOut              => lbInfoBitOutSgnExWire
        );


LdpccCodeSignExtractorIcfaalp2 : LdpccSignExtractor
  port map (
        clkClock              => clkClockWire,
        rstReset              => rstResetWire,
     llrduLlrDelayUnitIn   => llrduCodeLlrDelayUnitInSignExWire,
        --Outputs
```

```
        lbBitOut              => lbCodeBitOutSgnExWire
        );

OutputInterfaceIcfaalp2 : OutputInterface
    port map
        ( -- Inputs from Pins
          slvControlFp  => slvControlFp,
          -- Common Inputs
          clkClock      => clkClockWire,
          rstReset      => rstResetWire,
       -- Inputs from Encoder
          bitInfoBitInEnc  => bitInfoBitOutEncWire,
          bitCodeBitInEnc  => bitCodeBitOutEncWire,
          -- Inputs from Input Interface;
          llrduInfoLlrDelayUnitInIpIf  => llrduInfoLlrDelayUnitInOpIfWire,
          llrduCodeLlrDelayUnitInIpIf  => llrduCodeLlrDelayUnitInOpIfWire,
          phPhaseInIpIf                => phPhaseInOpIfWire,
          phaPhaseOffsetInIpIf         => phaPhaseOffsetInOpIfWire,
          -- Inputs from Processors
          llrduaInfoLlrDelayUnitInPro  => llrduaInfoLlrDelayUnitOutProWire,
          llrduaCodeLlrDelayUnitInPro  => llrduaCodeLlrDelayUnitOutProWire,
          -- Inputs from Sign Extractor
          lbInfoBitInSgnEx  => lbInfoBitOutSgnExWire,
          lbCodeBitInSgnEx  => lbCodeBitOutSgnExWire,
          -- Inputs from Error Counter
          slvBlockCountLog2InErr  => slvBlockCountLog2Wire,
          -- Outputs to Processor(s)
          phPhaseOutPro                 => phPhaseInProWire,

          llrduaInfoLlrDelayUnitOutPro => llrduaInfoLlrDelayUnitInProWire,
          llrduaCodeLlrDelayUnitOutPro => llrduaCodeLlrDelayUnitInProWire,
          phaPhaseOffsetOutPro         => phaPhaseOffsetInProWire,

          -- Outputs to Sign Extractors
          llrduInfoLleDelayUnitOutSignEx => llrduInfoLlrDelayUnitInSignExWire,
          llrduCodeLleDelayUnitOutSignEx => llrduCodeLlrDelayUnitInSignExWire,
          -- Outputs to Pins
          slvHighSpeedOutputsTp   =>      slvHighSpeedOutputsTp
          );

ErrorCounterIcfaalp2 :  ErrorCounter
  port map (
      -- inputs
      clkClock          => clkClockWire,
      rstReset          => rstResetWire,
      lbDecodedInfoBit => lbInfoBitOutSgnExWire,
      -- outputs
      slvBlockCountLog2      => slvBlockCountLog2Wire
      );


  -- We generate nNumProcessors processors using a generate statement and wire things
  -- up correctly.

   ProcessorGenerator : for nThisProcessor in 0 to nNumProcessors-1 generate
   begin
      LdpccProcessorIcfaalp2 : LdpccProcessor
    port map (
          clkClock                  => clkClockWire,
          rstReset                  => rstResetWire,
       llrduInfoLlrDelayUnitIn  => llrduaInfoLlrDelayUnitInProWire(nThisProcessor),
          llrduCodeLlrDelayUnitIn  => llrduaCodeLlrDelayUnitInProWire(nThisProcessor),
          phPhaseIn                => phPhaseInProWire,
          phPhaseOffset            => phaPhaseOffsetInProWire(nThisProcessor),
          --Outputs
          llrduInfoLlrDelayUnitOut => llrduaInfoLlrDelayUnitOutProWire(nThisProcessor),
          llrduCodeLlrDelayUnitOut => llrduaCodeLlrDelayUnitOutProWire(nThisProcessor)
        );
end generate ProcessorGenerator;

-- purpose: This process assigns the internal clock to an output pin.
-- type   : combinational
-- inputs : clkClockWire
-- outputs:
ASSIGN: process (clkClockWire)
begin  -- process ASSIGN
  clkClockTp <= clkClockWire;
end process ASSIGN;

 end architecture Behavioral;
```

Listing B.3: Icfaalp2InputInterface.vhd

```
--------------------------------------------------------------------------------
--
--    Filename: Icfaalp2InputInterface.vhd
--
--    Created:  29th April 2004
```

```
--    Version:   0.1
--
--    Author:    Stephen Bates & Ramkrishna Swamy
--    Location:  Dept. of Electrical and Computer Engineering
--               The University of Alberta
--               Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------------
--
--   Description
--   -----------
--
--   This is the VHDL file for the ICFAALP2 input interface which is to be
--   fabricated by TSMC 0.18u.
--
-- ------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity InputInterface is
    Port ( -- Inputs from Pins
           clkClockFp              : in std_logic;
           rstResetFp              : in std_logic;
           slvHighSpeedInputsFp    : in std_logic_vector ( NUM_HIGH_SPEED_IPS-1  downto 0 );
           slvControlFp            : in std_logic_vector ( NUM_CONTROL_BITS-1    downto 0 );
           -- Common Outputs
           clkClock                : out LogicBit;
           rstReset                : out LogicBit;
           --Outputs to Info Bit Gen
           slLoadGenOut       : out std_logic;
           slvSeedGenOut      : out std_logic_vector ( 31 downto 0 );
           slvTestPtnGenOut   : out std_logic_vector (  2 downto 0 );
           -- Inputs from Info Bit Gen
           slvInfoBitGenIn    : in std_logic;
           phPhaseGenIn       : in Phase;
           -- Outputs to LdpccEncoder
           bitInfoBitOutEnc   : out LogicBit;
           phPhaseOutEnc      : out Phase;
           -- Inputs from LdpccEncoder
           bitInfoBitInEnc    : in LogicBit;
           bitCodeBitInEnc    : in LogicBit;
           -- Outputs to Channel Emulator
           bitInfoBitOutChE      : out LogicBit;
           bitCodeBitOutChE      : out LogicBit;
           ctrlcheControlChE     : out std_logic_vector ( 2 downto 0 );
           -- Inputs from Channel Emulator
           llrInfoLlrInChE: in LogLikeRatio;
           llrCodeLlrInChE: in LogLikeRatio;
           -- Outputs to Output Interface
           llrduInfoLlrDelayUnitOutOpIf   : out LogLikeRatioDelayUnit;
           llrduCodeLlrDelayUnitOutOpIf   : out LogLikeRatioDelayUnit;
           phPhaseOutOpIf                 : out Phase;
           phaPhaseOffsetOutOpIf          : out PhaseArray
           );
end entity InputInterface;

architecture Behavioral of InputInterface is

signal phPhaseOutEnc_Del_Next : PhaseArray;
signal phPhaseOutEnc_Del      : PhaseArray;

signal slvHighSpeedInputsFp_Del1_Next : std_logic_vector ( NUM_HIGH_SPEED_IPS-1 downto 0);
signal slvHighSpeedInputsFp_Del1      : std_logic_vector ( NUM_HIGH_SPEED_IPS-1 downto 0);

begin

  -- This entity is simply a big mux which is driven by the control signal
  -- slvControlFp. Depending on its value certain inputs get driven to
  -- certain outputs.

-- ------------------------------------------------------------------------
--   DIRECT_ASSIGNMENTS
-- ------------------------------------------------------------------------

  -- This process always assigns certain signals to certain outputs regardless of the
  -- input control signal.

  DIRECT_ASSIGNMENTS : process ( clkClockFp,rstResetFp,slvControlFp ) is
  begin

    clkClock <= clkClockFp;
    rstReset <= rstResetFp;

    phaPhaseOffsetOutOpIf        <= PHASE_ARRAY_RESET;
```

```
    slLoadGenOut                <= '0';
    slvSeedGenOut               <= (others=>'0');
    slvTestPtnGenOut(2)         <= '0';
    slvTestPtnGenOut(1 downto 0) <= slvControlFp(3 downto 2);

  end process DIRECT_ASSIGNMENTS;

-------------------------------------------------------------------------------
-- ENCODER_INPUTS
-------------------------------------------------------------------------------

  -- Based on the control signals the inputs to the encoder can either come
  -- from off chip or from the LFSR (TBD).

  -- XXX...X00 => Encoder Inputs from off chip
  -- XXX...X01 => Encoder Inputs from LFSR (TBD)
  -- XXX...X1X => Encoder Inputs Zero'ed (helps test power).

  ENCODER_INPUTS : process ( slvControlFp, slvHighSpeedInputsFp_Del1, slvInfoBitGenIn,
                             phPhaseGenIn ) is
  begin

    if ( slvControlFp(1 downto 0) = "00" ) then
      bitInfoBitOutEnc          <= slvHighSpeedInputsFp_Del1(0);
      phPhaseOutEnc             <= slvHighSpeedInputsFp_Del1(8 downto 1);
      phPhaseOutEnc_Del_Next(0) <= slvHighSpeedInputsFp_Del1(8 downto 1);
    elsif ( slvControlFp(1 downto 0) = "01" ) then
      bitInfoBitOutEnc          <= slvInfoBitGenIn;
      phPhaseOutEnc             <= phPhaseGenIn;
      phPhaseOutEnc_Del_Next(0) <= phPhaseGenIn;
    else
      bitInfoBitOutEnc          <= '0';
      phPhaseOutEnc             <= (others=>'0');
      phPhaseOutEnc_Del_Next(0) <= (others=>'0');
    end if;

  end process ENCODER_INPUTS;

-------------------------------------------------------------------------------
-- CHANNEL_EMULATOR_INPUTS
-------------------------------------------------------------------------------

  -- The channel emulator inputs always come from the encoder outputs
  -- through this block. We also pass in a slice of the control signal
  -- to set the noise level.

  -- XXX..XAB0X => Encoder on, take noise value AB
  -- XXX..XXX1X => Encoder off, set noise to NOISE_OFF (see package for defn.)

  CHANNEL_EMULATOR_INPUTS : process ( bitInfoBitInEnc, bitCodeBitInEnc, slvControlFp ) is
  begin

    bitInfoBitOutChE <= bitInfoBitInEnc;
    bitCodeBitOutChE <= bitCodeBitInEnc;

    if ( slvControlFp(1) = '1' ) then
       ctrlcheControlChE <= NOISE_OFF;
    else
      ctrlcheControlChE(2 downto 0)  <= slvControlFp( 16 downto 14 );
    end if;

  end process CHANNEL_EMULATOR_INPUTS;

-------------------------------------------------------------------------------
-- DECODER_LLRS
-------------------------------------------------------------------------------

  -- The LLRs to be input from the encoder come either from off the chip (in which case
  -- we must reuse the phase and encoder input bits). Or from the channel emulator. This
  -- implies we can not use the pins to drive the encoder and decoder at the same
  -- time!!! This could change is we had enought pins.

  -- XXX...XX1X => Encoder is off so inputs must come from off chip.
  -- XXX...XX00 => Encoder on with inputs from off chip. Decoder LLRs come
  --               from channel emulator and phase is delayed version of
  --               what came from the pins.
  -- XXX...XX01 => Encoder on with inputs from LFSR. Decoder LLRs come from
  --               channel emulator with phase delayed version of that driving
  --               encoder (simply a counter from reset).

  DECODER_LLRS : process ( slvControlFp, slvHighSpeedInputsFp_Del1, llrInfoLlrInChE,
                           llrCodeLlrInChE, phPhaseOutEnc_Del ) is
  begin

    if ( slvControlFp(1) = '1' ) then

    llrduInfoLlrDelayUnitOutOpIf(0) <= (sat=>slvHighSpeedInputsFp_Del1((nLlrWidth+2)-1) , Sgn=> →
        ↪ slvHighSpeedInputsFp_Del1((nLlrWidth+2)-2) ,
          Mag=>slvHighSpeedInputsFp_Del1((nLlrWidth+2)-3 downto (nLlrWidth+2)-nLlrWidth-2) );
```

```
    llrduInfoLlrDelayUnitOutOpIf(1) <= (sat=>slvHighSpeedInputsFp_Del1((nLlrWidth+2)-1) , Sgn=> →
        ↪ slvHighSpeedInputsFp_Del1((nLlrWidth+2)-2) ,
          Mag=>slvHighSpeedInputsFp_Del1((nLlrWidth+2)-3 downto (nLlrWidth+2)-nLlrWidth -2) );
    llrduInfoLlrDelayUnitOutOpIf(2) <= (sat=>slvHighSpeedInputsFp_Del1((nLlrWidth+2)-1) , Sgn=> →
        ↪ slvHighSpeedInputsFp_Del1((nLlrWidth+2)-2) ,
          Mag=>slvHighSpeedInputsFp_Del1((nLlrWidth+2)-3 downto (nLlrWidth+2)-nLlrWidth -2) );
    llrduInfoLlrDelayUnitOutOpIf(3) <= (sat=>slvHighSpeedInputsFp_Del1((nLlrWidth+2)-1) , Sgn=> →
        ↪ slvHighSpeedInputsFp_Del1((nLlrWidth+2)-2) ,
          Mag=>slvHighSpeedInputsFp_Del1((nLlrWidth+2)-3 downto (nLlrWidth+2)-nLlrWidth -2) );


    llrduCodeLlrDelayUnitOutOpIf(0) <= (sat=>slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-1) , Sgn=> →
        ↪ slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-2) ,
          Mag=>slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-3 downto 2*(nLlrWidth+2)-nLlrWidth -2) );
    llrduCodeLlrDelayUnitOutOpIf(1) <= (sat=>slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-1) , Sgn=> →
        ↪ slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-2) ,
          Mag=>slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-3 downto 2*(nLlrWidth+2)-nLlrWidth -2) );
    llrduCodeLlrDelayUnitOutOpIf(2) <= (sat=>slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-1) , Sgn=> →
        ↪ slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-2) ,
          Mag=>slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-3 downto 2*(nLlrWidth+2)-nLlrWidth -2) );
    llrduCodeLlrDelayUnitOutOpIf(3) <= (sat=>slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-1) , Sgn=> →
        ↪ slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-2) ,
          Mag=>slvHighSpeedInputsFp_Del1(2*(nLlrWidth+2)-3 downto 2*(nLlrWidth+2)-nLlrWidth -2) );

    phPhaseOutOpIf  <= slvHighSpeedInputsFp_Del1( 2*(nLlrWidth+2)-1+8 downto 2*(nLlrWidth+2) );

    else

    llrduInfoLlrDelayUnitOutOpIf(0) <= llrInfoLlrInChE;
    llrduInfoLlrDelayUnitOutOpIf(1) <= llrInfoLlrInChE;
    llrduInfoLlrDelayUnitOutOpIf(2) <= llrInfoLlrInChE;
    llrduInfoLlrDelayUnitOutOpIf(3) <= llrInfoLlrInChE;
    llrduCodeLlrDelayUnitOutOpIf(0) <= llrCodeLlrInChE;
    llrduCodeLlrDelayUnitOutOpIf(1) <= llrCodeLlrInChE;
    llrduCodeLlrDelayUnitOutOpIf(2) <= llrCodeLlrInChE;
    llrduCodeLlrDelayUnitOutOpIf(3) <= llrCodeLlrInChE;

    phPhaseOutOpIf  <= phPhaseOutEnc_Del(PHASE_IN_DELAY);

    end if;
  end process DECODER_LLRS;

-------------------------------------------------------------------------------
-- PHASE_DELAY_LINE
-------------------------------------------------------------------------------
  -- This process

  PHASE_DELAY_LINE : process ( phPhaseOutEnc_Del ) is
  begin

  phPhaseOutEnc_Del_Next(phPhaseOutEnc_Del_Next'left(1) downto 1) <=
          phPhaseOutEnc_Del(phPhaseOutEnc_Del_Next'left(1)-1 downto 0);

  end process PHASE_DELAY_LINE;
-------------------------------------------------------------------------------
-- HIGH_IN_DELAY
-------------------------------------------------------------------------------
  -- This process

  HIGH_IN_DELAY : process ( slvHighSpeedInputsFp ) is
  begin

  slvHighSpeedInputsFp_Del1_Next <= slvHighSpeedInputsFp;

  end process HIGH_IN_DELAY;
-------------------------------------------------------------------------------
-- CLOCK_UPDATE
-------------------------------------------------------------------------------
    -- This is the process which updates all the registers
    -- with their new values.

  CLOCK_UPDATE : process ( clkClockFp ) is
  begin

    if ( clkClockFp'event and clkClockFp='1' ) then
      if ( rstResetFp = '1' ) then
        phPhaseOutEnc_Del          <= (others=>(others=>'0'));
        slvHighSpeedInputsFp_Del1 <= (others=>'0');
      else
        phPhaseOutEnc_Del          <= phPhaseOutEnc_Del_Next;
        slvHighSpeedInputsFp_Del1 <= slvHighSpeedInputsFp_Del1_Next;
      end if;
    end if;
  end process CLOCK_UPDATE;

end Behavioral;
```

Listing B.4: LdpccEncoder.vhd

```
--
--    Filename: LdpccEncoder.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------
--
--   Description
--   -----------
--
--  This is the encoder block for the LDPC-CC.
--
-- ------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity LdpccEncoder is
    Port ( clkClock       : in LogicBit;
           rstReset       : in LogicBit;
           bitInfoBitIn   : in LogicBit;
           phPhaseIn      : in Phase;
           --Outputs
           bitInfoBitOut  : out LogicBit;
           bitCodeBitOut  : out LogicBit
         );
end entity LdpccEncoder;

architecture Behavioral of LdpccEncoder is

    -- Registered values which must have an _Next value
    -- to work well. As per Massana design rules.

   signal emInfoBitDelayLine_Next : EncoderMemory;
   signal emInfoBitDelayLine      : EncoderMemory;

   signal emCodeBitDelayLine_Next : EncoderMemory;
   signal emCodeBitDelayLine      : EncoderMemory;

   signal bitInfoBitIn_Del1_Next : LogicBit;
   signal bitInfoBitIn_Del1      : LogicBit;

   signal phPhaseIn_Del1_Next : Phase;
   signal phPhaseIn_Del1      : Phase;


    -- Combinatorial signals that are not directly assigned to a
    -- register.

   signal bitCodeBit      : LogicBit;

begin

    -- Update the _Next values of the delay lines based on the
    -- new inputs and the new code bit.

   DELAY_UPDATE : process ( bitInfoBitIn_Del1 , bitCodeBit , emInfoBitDelayLine , emCodeBitDelayLine ,
                            bitInfoBitIn , phPhaseIn ) is
   begin


        -- Move all the delay line elements up from the registered
        -- counterparts.
       emInfoBitDelayLine_Next(emInfoBitDelayLine'left(1) downto 1) <=
             emInfoBitDelayLine(emInfoBitDelayLine'left(1)-1 downto 0);
       emCodeBitDelayLine_Next(emCodeBitDelayLine'left(1) downto 1) <=
             emCodeBitDelayLine(emCodeBitDelayLine'left(1)-1 downto 0);

        -- Place the most recent Information bit and code bit into
        -- first point of the delay line (entry 0)
       emInfoBitDelayLine_Next(0) <= bitInfoBitIn_Del1;
```

```
        emCodeBitDelayLine_Next(0) <= bitCodeBit;

          --Register the inputs to aid synthesis.
        phPhaseIn_Del1_Next     <= phPhaseIn;
        bitInfoBitIn_Del1_Next <= bitInfoBitIn;

end process DELAY_UPDATE;

  -- Perform the XOR operation to determine the new parity (code)
  -- bit.

PARITY_GEN : process ( phPhaseIn_Del1 ,emCodeBitDelayLine ,emInfoBitDelayLine ,bitInfoBitIn_Del1 ) is
begin

    --Start of Auto-generated by VhdlForLdpccEncoder.m see sbates for details.
  case phPhaseIn_Del1 is
  when "00000000" =>
      bitCodeBit <= emInfoBitDelayLine(7) xor emCodeBitDelayLine(10) xor emInfoBitDelayLine(64) xor →
          ↪ emInfoBitDelayLine(104) xor emCodeBitDelayLine(127);
  when "00000001" =>
      bitCodeBit <= emCodeBitDelayLine(8) xor emInfoBitDelayLine(31) xor emInfoBitDelayLine(52) xor →
          ↪ emInfoBitDelayLine(120) xor emCodeBitDelayLine(127);
  when "00000010" =>
      bitCodeBit <= emCodeBitDelayLine(28) xor emInfoBitDelayLine(39) xor emInfoBitDelayLine(61) xor →
          ↪ emInfoBitDelayLine(107) xor emCodeBitDelayLine(127);
  when "00000011" =>
      bitCodeBit <= emInfoBitDelayLine(11) xor emCodeBitDelayLine(19) xor emInfoBitDelayLine(72) xor →
          ↪ emInfoBitDelayLine(102) xor emCodeBitDelayLine(127);
  when "00000100" =>
      bitCodeBit <= emInfoBitDelayLine(24) xor emCodeBitDelayLine(37) xor emInfoBitDelayLine(56) xor →
          ↪ emInfoBitDelayLine(112) xor emCodeBitDelayLine(127);
  when "00000101" =>
      bitCodeBit <= emInfoBitDelayLine(8) xor emCodeBitDelayLine(27) xor emInfoBitDelayLine(76) xor →
          ↪ emInfoBitDelayLine(114) xor emCodeBitDelayLine(127);
  when "00000110" =>
      bitCodeBit <= emCodeBitDelayLine(21) xor emInfoBitDelayLine(37) xor emInfoBitDelayLine(59) xor →
          ↪ emInfoBitDelayLine(124) xor emCodeBitDelayLine(127);
  when "00000111" =>
      bitCodeBit <= emInfoBitDelayLine(1) xor emInfoBitDelayLine(29) xor emCodeBitDelayLine(30) xor →
          ↪ emInfoBitDelayLine(69) xor emCodeBitDelayLine(127);
  when "00001000" =>
      bitCodeBit <= emInfoBitDelayLine(21) xor emCodeBitDelayLine(33) xor emInfoBitDelayLine(78) xor →
          ↪ emInfoBitDelayLine(94) xor emCodeBitDelayLine(127);
  when "00001001" =>
      bitCodeBit <= emInfoBitDelayLine(23) xor emCodeBitDelayLine(38) xor emInfoBitDelayLine(54) xor →
          ↪ emInfoBitDelayLine(125) xor emCodeBitDelayLine(127);
  when "00001010" =>
      bitCodeBit <= emCodeBitDelayLine(41) xor emInfoBitDelayLine(51) xor emInfoBitDelayLine(73) xor →
          ↪ emInfoBitDelayLine(124) xor emCodeBitDelayLine(127);
  when "00001011" =>
      bitCodeBit <= emInfoBitDelayLine(10) xor emCodeBitDelayLine(14) xor emInfoBitDelayLine(51) xor →
          ↪ emInfoBitDelayLine(88) xor emCodeBitDelayLine(127);
  when "00001100" =>
      bitCodeBit <= emInfoBitDelayLine(18) xor emCodeBitDelayLine(24) xor emInfoBitDelayLine(67) xor →
          ↪ emInfoBitDelayLine(115) xor emCodeBitDelayLine(127);
  when "00001101" =>
      bitCodeBit <= emInfoBitDelayLine(6) xor emCodeBitDelayLine(21) xor emInfoBitDelayLine(32) xor →
          ↪ emInfoBitDelayLine(80) xor emCodeBitDelayLine(127);
  when "00001110" =>
      bitCodeBit <= emCodeBitDelayLine(14) xor emInfoBitDelayLine(40) xor emInfoBitDelayLine(87) xor →
          ↪ emInfoBitDelayLine(110) xor emCodeBitDelayLine(127);
  when "00001111" =>
      bitCodeBit <= emInfoBitDelayLine(43) xor emCodeBitDelayLine(50) xor emInfoBitDelayLine(94) xor →
          ↪ emInfoBitDelayLine(125) xor emCodeBitDelayLine(127);
  when "00010000" =>
      bitCodeBit <= emCodeBitDelayLine(30) xor emInfoBitDelayLine(34) xor emInfoBitDelayLine(98) xor →
          ↪ emInfoBitDelayLine(118) xor emCodeBitDelayLine(127);
  when "00010001" =>
      bitCodeBit <= emCodeBitDelayLine(34) xor emInfoBitDelayLine(34) xor emInfoBitDelayLine(77) xor →
          ↪ emInfoBitDelayLine(108) xor emCodeBitDelayLine(127);
  when "00010010" =>
      bitCodeBit <= emCodeBitDelayLine(20) xor emInfoBitDelayLine(42) xor emInfoBitDelayLine(96) xor →
          ↪ emInfoBitDelayLine(124) xor emCodeBitDelayLine(127);
  when "00010011" =>
      bitCodeBit <= emInfoBitDelayLine(20) xor emCodeBitDelayLine(57) xor emInfoBitDelayLine(65) xor →
          ↪ emInfoBitDelayLine(109) xor emCodeBitDelayLine(127);
  when "00010100" =>
      bitCodeBit <= emInfoBitDelayLine(6) xor emInfoBitDelayLine(24) xor emCodeBitDelayLine(38) xor →
          ↪ emInfoBitDelayLine(77) xor emCodeBitDelayLine(127);
  when "00010101" =>
      bitCodeBit <= bitInfoBitIn_Del1 xor emCodeBitDelayLine(42) xor emInfoBitDelayLine(50) xor →
          ↪ emInfoBitDelayLine(89) xor emCodeBitDelayLine(127);
  when "00010110" =>
      bitCodeBit <= emInfoBitDelayLine(27) xor emCodeBitDelayLine(63) xor emInfoBitDelayLine(76) xor →
          ↪ emInfoBitDelayLine(119) xor emCodeBitDelayLine(127);
  when "00010111" =>
      bitCodeBit <= emCodeBitDelayLine(43) xor emInfoBitDelayLine(48) xor emInfoBitDelayLine(99) xor →
          ↪ emInfoBitDelayLine(124) xor emCodeBitDelayLine(127);
  when "00011000" =>
```

```
    bitCodeBit <= emCodeBitDelayLine(35) xor emInfoBitDelayLine(39) xor emInfoBitDelayLine(105) xor →
        ↪ emInfoBitDelayLine(119) xor emCodeBitDelayLine(127);
when "00011001" =>
    bitCodeBit <= emInfoBitDelayLine(22) xor emCodeBitDelayLine(52) xor emInfoBitDelayLine(67) xor →
        ↪ emInfoBitDelayLine(86) xor emCodeBitDelayLine(127);
when "00011010" =>
    bitCodeBit <= emInfoBitDelayLine(49) xor emCodeBitDelayLine(62) xor emInfoBitDelayLine(92) xor →
        ↪ emInfoBitDelayLine(124) xor emCodeBitDelayLine(127);
when "00011011" =>
    bitCodeBit <= emCodeBitDelayLine(61) xor emInfoBitDelayLine(61) xor emInfoBitDelayLine(85) xor →
        ↪ emInfoBitDelayLine(114) xor emCodeBitDelayLine(127);
when "00011100" =>
    bitCodeBit <= emInfoBitDelayLine(11) xor emInfoBitDelayLine(38) xor emCodeBitDelayLine(67) xor →
        ↪ emInfoBitDelayLine(93) xor emCodeBitDelayLine(127);
when "00011101" =>
    bitCodeBit <= emInfoBitDelayLine(13) xor emInfoBitDelayLine(41) xor emCodeBitDelayLine(48) xor →
        ↪ emInfoBitDelayLine(76) xor emCodeBitDelayLine(127);
when "00011110" =>
    bitCodeBit <= emInfoBitDelayLine(32) xor emCodeBitDelayLine(62) xor emInfoBitDelayLine(104) xor →
        ↪ emInfoBitDelayLine(122) xor emCodeBitDelayLine(127);
when "00011111" =>
    bitCodeBit <= emInfoBitDelayLine(23) xor emInfoBitDelayLine(66) xor emCodeBitDelayLine(71) xor →
        ↪ emInfoBitDelayLine(87) xor emCodeBitDelayLine(127);
when "00100000" =>
    bitCodeBit <= emInfoBitDelayLine(28) xor emCodeBitDelayLine(60) xor emInfoBitDelayLine(68) xor →
        ↪ emInfoBitDelayLine(116) xor emCodeBitDelayLine(127);
when "00100001" =>
    bitCodeBit <= emInfoBitDelayLine(24) xor emInfoBitDelayLine(33) xor emCodeBitDelayLine(75) xor →
        ↪ emInfoBitDelayLine(113) xor emCodeBitDelayLine(127);
when "00100010" =>
    bitCodeBit <= emCodeBitDelayLine(43) xor emInfoBitDelayLine(50) xor emInfoBitDelayLine(119) xor →
        ↪ emInfoBitDelayLine(122) xor emCodeBitDelayLine(127);
when "00100011" =>
    bitCodeBit <= emInfoBitDelayLine(30) xor emCodeBitDelayLine(59) xor emInfoBitDelayLine(67) xor →
        ↪ emInfoBitDelayLine(79) xor emCodeBitDelayLine(127);
when "00100100" =>
    bitCodeBit <= emInfoBitDelayLine(7) xor emCodeBitDelayLine(42) xor emInfoBitDelayLine(75) xor →
        ↪ emInfoBitDelayLine(108) xor emCodeBitDelayLine(127);
when "00100101" =>
    bitCodeBit <= emInfoBitDelayLine(19) xor emInfoBitDelayLine(46) xor emCodeBitDelayLine(74) xor →
        ↪ emInfoBitDelayLine(85) xor emCodeBitDelayLine(127);
when "00100110" =>
    bitCodeBit <= emInfoBitDelayLine(2) xor emCodeBitDelayLine(43) xor emInfoBitDelayLine(76) xor →
        ↪ emInfoBitDelayLine(87) xor emCodeBitDelayLine(127);
when "00100111" =>
    bitCodeBit <= bitInfoBitIn_Del1 xor emCodeBitDelayLine(52) xor emInfoBitDelayLine(60) xor →
        ↪ emInfoBitDelayLine(122) xor emCodeBitDelayLine(127);
when "00101000" =>
    bitCodeBit <= emInfoBitDelayLine(5) xor emCodeBitDelayLine(70) xor emInfoBitDelayLine(73) xor →
        ↪ emInfoBitDelayLine(90) xor emCodeBitDelayLine(127);
when "00101001" =>
    bitCodeBit <= emInfoBitDelayLine(29) xor emCodeBitDelayLine(45) xor emInfoBitDelayLine(52) xor →
        ↪ emInfoBitDelayLine(84) xor emCodeBitDelayLine(127);
when "00101010" =>
    bitCodeBit <= emInfoBitDelayLine(40) xor emCodeBitDelayLine(43) xor emInfoBitDelayLine(69) xor →
        ↪ emInfoBitDelayLine(117) xor emCodeBitDelayLine(127);
when "00101011" =>
    bitCodeBit <= emCodeBitDelayLine(26) xor emInfoBitDelayLine(26) xor emInfoBitDelayLine(46) xor →
        ↪ emInfoBitDelayLine(110) xor emCodeBitDelayLine(127);
when "00101100" =>
    bitCodeBit <= emInfoBitDelayLine(1) xor emCodeBitDelayLine(36) xor emInfoBitDelayLine(44) xor →
        ↪ emInfoBitDelayLine(121) xor emCodeBitDelayLine(127);
when "00101101" =>
    bitCodeBit <= emCodeBitDelayLine(10) xor emInfoBitDelayLine(32) xor emInfoBitDelayLine(69) xor →
        ↪ emInfoBitDelayLine(91) xor emCodeBitDelayLine(127);
when "00101110" =>
    bitCodeBit <= emInfoBitDelayLine(1) xor emCodeBitDelayLine(10) xor emInfoBitDelayLine(45) xor →
        ↪ emInfoBitDelayLine(79) xor emCodeBitDelayLine(127);
when "00101111" =>
    bitCodeBit <= emInfoBitDelayLine(9) xor emCodeBitDelayLine(40) xor emInfoBitDelayLine(56) xor →
        ↪ emInfoBitDelayLine(97) xor emCodeBitDelayLine(127);
when "00110000" =>
    bitCodeBit <= emCodeBitDelayLine(14) xor emInfoBitDelayLine(39) xor emInfoBitDelayLine(63) xor →
        ↪ emInfoBitDelayLine(97) xor emCodeBitDelayLine(127);
when "00110001" =>
    bitCodeBit <= emInfoBitDelayLine(28) xor emCodeBitDelayLine(31) xor emInfoBitDelayLine(53) xor →
        ↪ emInfoBitDelayLine(113) xor emCodeBitDelayLine(127);
when "00110010" =>
    bitCodeBit <= emCodeBitDelayLine(13) xor emInfoBitDelayLine(23) xor emInfoBitDelayLine(57) xor →
        ↪ emInfoBitDelayLine(125) xor emCodeBitDelayLine(127);
when "00110011" =>
    bitCodeBit <= emInfoBitDelayLine(16) xor emCodeBitDelayLine(47) xor emInfoBitDelayLine(56) xor →
        ↪ emInfoBitDelayLine(109) xor emCodeBitDelayLine(127);
when "00110100" =>
    bitCodeBit <= emInfoBitDelayLine(29) xor emCodeBitDelayLine(46) xor emInfoBitDelayLine(83) xor →
        ↪ emInfoBitDelayLine(99) xor emCodeBitDelayLine(127);
when "00110101" =>
    bitCodeBit <= emInfoBitDelayLine(29) xor emCodeBitDelayLine(52) xor emInfoBitDelayLine(67) xor →
        ↪ emInfoBitDelayLine(115) xor emCodeBitDelayLine(127);
```

```
when "00110110" =>
    bitCodeBit <= emInfoBitDelayLine(18) xor emCodeBitDelayLine(32) xor emInfoBitDelayLine(80) xor →
        ↪ emInfoBitDelayLine(120) xor emCodeBitDelayLine(127);
when "00110111" =>
    bitCodeBit <= emInfoBitDelayLine(5) xor emInfoBitDelayLine(13) xor emCodeBitDelayLine(27) xor →
        ↪ emInfoBitDelayLine(71) xor emCodeBitDelayLine(127);
when "00111000" =>
    bitCodeBit <= emInfoBitDelayLine(9) xor emCodeBitDelayLine(33) xor emInfoBitDelayLine(34) xor →
        ↪ emInfoBitDelayLine(81) xor emCodeBitDelayLine(127);
when "00111001" =>
    bitCodeBit <= emInfoBitDelayLine(17) xor emCodeBitDelayLine(52) xor emInfoBitDelayLine(92) xor →
        ↪ emInfoBitDelayLine(116) xor emCodeBitDelayLine(127);
when "00111010" =>
    bitCodeBit <= emInfoBitDelayLine(29) xor emCodeBitDelayLine(32) xor emInfoBitDelayLine(96) xor →
        ↪ emInfoBitDelayLine(113) xor emCodeBitDelayLine(127);
when "00111011" =>
    bitCodeBit <= emInfoBitDelayLine(22) xor emCodeBitDelayLine(34) xor emInfoBitDelayLine(87) xor →
        ↪ emInfoBitDelayLine(124) xor emCodeBitDelayLine(127);
when "00111100" =>
    bitCodeBit <= emInfoBitDelayLine(0) xor emInfoBitDelayLine(41) xor emCodeBitDelayLine(50) xor →
        ↪ emInfoBitDelayLine(102) xor emCodeBitDelayLine(127);
when "00111101" =>
    bitCodeBit <= emCodeBitDelayLine(48) xor emInfoBitDelayLine(57) xor emInfoBitDelayLine(88) xor →
        ↪ emInfoBitDelayLine(106) xor emCodeBitDelayLine(127);
when "00111110" =>
    bitCodeBit <= emCodeBitDelayLine(19) xor emInfoBitDelayLine(30) xor emInfoBitDelayLine(74) xor →
        ↪ emInfoBitDelayLine(122) xor emCodeBitDelayLine(127);
when "00111111" =>
    bitCodeBit <= emInfoBitDelayLine(29) xor emCodeBitDelayLine(48) xor emInfoBitDelayLine(76) xor →
        ↪ emInfoBitDelayLine(111) xor emCodeBitDelayLine(127);
when "01000000" =>
    bitCodeBit <= emInfoBitDelayLine(6) xor emCodeBitDelayLine(43) xor emInfoBitDelayLine(58) xor →
        ↪ emInfoBitDelayLine(98) xor emCodeBitDelayLine(127);
when "01000001" =>
    bitCodeBit <= emInfoBitDelayLine(16) xor emCodeBitDelayLine(35) xor emInfoBitDelayLine(37) xor →
        ↪ emInfoBitDelayLine(97) xor emCodeBitDelayLine(127);
when "01000010" =>
    bitCodeBit <= emCodeBitDelayLine(57) xor emInfoBitDelayLine(59) xor emInfoBitDelayLine(105) xor →
        ↪ emInfoBitDelayLine(117) xor emCodeBitDelayLine(127);
when "01000011" =>
    bitCodeBit <= emCodeBitDelayLine(40) xor emInfoBitDelayLine(64) xor emInfoBitDelayLine(88) xor →
        ↪ emInfoBitDelayLine(123) xor emCodeBitDelayLine(127);
when "01000100" =>
    bitCodeBit <= emInfoBitDelayLine(17) xor emCodeBitDelayLine(39) xor emInfoBitDelayLine(58) xor →
        ↪ emInfoBitDelayLine(108) xor emCodeBitDelayLine(127);
when "01000101" =>
    bitCodeBit <= emInfoBitDelayLine(14) xor emInfoBitDelayLine(49) xor emCodeBitDelayLine(53) xor →
        ↪ emInfoBitDelayLine(99) xor emCodeBitDelayLine(127);
when "01000110" =>
    bitCodeBit <= emCodeBitDelayLine(29) xor emInfoBitDelayLine(29) xor emInfoBitDelayLine(71) xor →
        ↪ emInfoBitDelayLine(114) xor emCodeBitDelayLine(127);
when "01000111" =>
    bitCodeBit <= emCodeBitDelayLine(33) xor emInfoBitDelayLine(41) xor emInfoBitDelayLine(77) xor →
        ↪ emInfoBitDelayLine(123) xor emCodeBitDelayLine(127);
when "01001000" =>
    bitCodeBit <= emInfoBitDelayLine(41) xor emCodeBitDelayLine(70) xor emInfoBitDelayLine(108) xor →
        ↪ emInfoBitDelayLine(115) xor emCodeBitDelayLine(127);
when "01001001" =>
    bitCodeBit <= emCodeBitDelayLine(41) xor emInfoBitDelayLine(65) xor emInfoBitDelayLine(102) xor →
        ↪ emInfoBitDelayLine(126) xor emCodeBitDelayLine(127);
when "01001010" =>
    bitCodeBit <= emInfoBitDelayLine(15) xor emCodeBitDelayLine(41) xor emInfoBitDelayLine(60) xor →
        ↪ emInfoBitDelayLine(85) xor emCodeBitDelayLine(127);
when "01001011" =>
    bitCodeBit <= emInfoBitDelayLine(3) xor emInfoBitDelayLine(42) xor emCodeBitDelayLine(72) xor →
        ↪ emInfoBitDelayLine(83) xor emCodeBitDelayLine(127);
when "01001100" =>
    bitCodeBit <= emInfoBitDelayLine(23) xor emCodeBitDelayLine(34) xor emInfoBitDelayLine(50) xor →
        ↪ emInfoBitDelayLine(98) xor emCodeBitDelayLine(127);
when "01001101" =>
    bitCodeBit <= emInfoBitDelayLine(2) xor emCodeBitDelayLine(57) xor emInfoBitDelayLine(72) xor →
        ↪ emInfoBitDelayLine(79) xor emCodeBitDelayLine(127);
when "01001110" =>
    bitCodeBit <= emInfoBitDelayLine(21) xor emCodeBitDelayLine(47) xor emInfoBitDelayLine(53) xor →
        ↪ emInfoBitDelayLine(115) xor emCodeBitDelayLine(127);
when "01001111" =>
    bitCodeBit <= emInfoBitDelayLine(33) xor emInfoBitDelayLine(67) xor emCodeBitDelayLine(68) xor →
        ↪ emInfoBitDelayLine(99) xor emCodeBitDelayLine(127);
when "01010000" =>
    bitCodeBit <= emInfoBitDelayLine(36) xor emCodeBitDelayLine(61) xor emInfoBitDelayLine(64) xor →
        ↪ emInfoBitDelayLine(97) xor emCodeBitDelayLine(127);
when "01010001" =>
    bitCodeBit <= emInfoBitDelayLine(20) xor emInfoBitDelayLine(66) xor emCodeBitDelayLine(69) xor →
        ↪ emInfoBitDelayLine(100) xor emCodeBitDelayLine(127);
when "01010010" =>
    bitCodeBit <= emInfoBitDelayLine(26) xor emCodeBitDelayLine(43) xor emInfoBitDelayLine(64) xor →
        ↪ emInfoBitDelayLine(105) xor emCodeBitDelayLine(127);
when "01010011" =>
```

```
        bitCodeBit <= emInfoBitDelayLine(15) xor emInfoBitDelayLine(44) xor emCodeBitDelayLine(59) xor →
            ↪ emInfoBitDelayLine(93) xor emCodeBitDelayLine(127);
    when "01010100" =>
        bitCodeBit <= emInfoBitDelayLine(18) xor emCodeBitDelayLine(44) xor emInfoBitDelayLine(73) xor →
            ↪ emInfoBitDelayLine(102) xor emCodeBitDelayLine(127);
    when "01010101" =>
        bitCodeBit <= emInfoBitDelayLine(37) xor emCodeBitDelayLine(71) xor emInfoBitDelayLine(83) xor →
            ↪ emInfoBitDelayLine(126) xor emCodeBitDelayLine(127);
    when "01010110" =>
        bitCodeBit <= emCodeBitDelayLine(17) xor emInfoBitDelayLine(40) xor emInfoBitDelayLine(62) xor →
            ↪ emInfoBitDelayLine(118) xor emCodeBitDelayLine(127);
    when "01010111" =>
        bitCodeBit <= emCodeBitDelayLine(22) xor emInfoBitDelayLine(39) xor emInfoBitDelayLine(65) xor →
            ↪ emInfoBitDelayLine(96) xor emCodeBitDelayLine(127);
    when "01011000" =>
        bitCodeBit <= emInfoBitDelayLine(7) xor emCodeBitDelayLine(32) xor emInfoBitDelayLine(69) xor →
            ↪ emInfoBitDelayLine(112) xor emCodeBitDelayLine(127);
    when "01011001" =>
        bitCodeBit <= emCodeBitDelayLine(8) xor emInfoBitDelayLine(20) xor emInfoBitDelayLine(77) xor →
            ↪ emInfoBitDelayLine(123) xor emCodeBitDelayLine(127);
    when "01011010" =>
        bitCodeBit <= emInfoBitDelayLine(17) xor emCodeBitDelayLine(42) xor emInfoBitDelayLine(61) xor →
            ↪ emInfoBitDelayLine(106) xor emCodeBitDelayLine(127);
    when "01011011" =>
        bitCodeBit <= emInfoBitDelayLine(25) xor emCodeBitDelayLine(39) xor emInfoBitDelayLine(82) xor →
            ↪ emInfoBitDelayLine(95) xor emCodeBitDelayLine(127);
    when "01011100" =>
        bitCodeBit <= emInfoBitDelayLine(7) xor emCodeBitDelayLine(12) xor emInfoBitDelayLine(60) xor →
            ↪ emInfoBitDelayLine(92) xor emCodeBitDelayLine(127);
    when "01011101" =>
        bitCodeBit <= emInfoBitDelayLine(25) xor emCodeBitDelayLine(42) xor emInfoBitDelayLine(86) xor →
            ↪ emInfoBitDelayLine(108) xor emCodeBitDelayLine(127);
    when "01011110" =>
        bitCodeBit <= emInfoBitDelayLine(4) xor emCodeBitDelayLine(30) xor emInfoBitDelayLine(30) xor →
            ↪ emInfoBitDelayLine(79) xor emCodeBitDelayLine(127);
    when "01011111" =>
        bitCodeBit <= emInfoBitDelayLine(9) xor emCodeBitDelayLine(12) xor emInfoBitDelayLine(87) xor →
            ↪ emInfoBitDelayLine(102) xor emCodeBitDelayLine(127);
    when "01100000" =>
        bitCodeBit <= emCodeBitDelayLine(10) xor emInfoBitDelayLine(19) xor emInfoBitDelayLine(90) xor →
            ↪ emInfoBitDelayLine(109) xor emCodeBitDelayLine(127);
    when "01100001" =>
        bitCodeBit <= emInfoBitDelayLine(42) xor emCodeBitDelayLine(43) xor emInfoBitDelayLine(84) xor →
            ↪ emInfoBitDelayLine(102) xor emCodeBitDelayLine(127);
    when "01100010" =>
        bitCodeBit <= emInfoBitDelayLine(24) xor emCodeBitDelayLine(45) xor emInfoBitDelayLine(67) xor →
            ↪ emInfoBitDelayLine(119) xor emCodeBitDelayLine(127);
    when "01100011" =>
        bitCodeBit <= emInfoBitDelayLine(34) xor emCodeBitDelayLine(40) xor emInfoBitDelayLine(64) xor →
            ↪ emInfoBitDelayLine(105) xor emCodeBitDelayLine(127);
    when "01100100" =>
        bitCodeBit <= emInfoBitDelayLine(0) xor emCodeBitDelayLine(28) xor emInfoBitDelayLine(48) xor →
            ↪ emInfoBitDelayLine(70) xor emCodeBitDelayLine(127);
    when "01100101" =>
        bitCodeBit <= emInfoBitDelayLine(26) xor emCodeBitDelayLine(30) xor emInfoBitDelayLine(80) xor →
            ↪ emInfoBitDelayLine(118) xor emCodeBitDelayLine(127);
    when "01100110" =>
        bitCodeBit <= emInfoBitDelayLine(15) xor emInfoBitDelayLine(30) xor emCodeBitDelayLine(55) xor →
            ↪ emInfoBitDelayLine(92) xor emCodeBitDelayLine(127);
    when "01100111" =>
        bitCodeBit <= emCodeBitDelayLine(24) xor emInfoBitDelayLine(43) xor emInfoBitDelayLine(62) xor →
            ↪ emInfoBitDelayLine(104) xor emCodeBitDelayLine(127);
    when "01101000" =>
        bitCodeBit <= emInfoBitDelayLine(55) xor emCodeBitDelayLine(60) xor emInfoBitDelayLine(70) xor →
            ↪ emInfoBitDelayLine(118) xor emCodeBitDelayLine(127);
    when "01101001" =>
        bitCodeBit <= emCodeBitDelayLine(29) xor emInfoBitDelayLine(51) xor emInfoBitDelayLine(91) xor →
            ↪ emInfoBitDelayLine(115) xor emCodeBitDelayLine(127);
    when "01101010" =>
        bitCodeBit <= emInfoBitDelayLine(12) xor emInfoBitDelayLine(23) xor emCodeBitDelayLine(48) xor →
            ↪ emInfoBitDelayLine(81) xor emCodeBitDelayLine(127);
    when "01101011" =>
        bitCodeBit <= emInfoBitDelayLine(9) xor emCodeBitDelayLine(32) xor emInfoBitDelayLine(49) xor →
            ↪ emInfoBitDelayLine(103) xor emCodeBitDelayLine(127);
    when "01101100" =>
        bitCodeBit <= emInfoBitDelayLine(4) xor emCodeBitDelayLine(34) xor emInfoBitDelayLine(57) xor →
            ↪ emInfoBitDelayLine(92) xor emCodeBitDelayLine(127);
    when "01101101" =>
        bitCodeBit <= emInfoBitDelayLine(7) xor emCodeBitDelayLine(49) xor emInfoBitDelayLine(62) xor →
            ↪ emInfoBitDelayLine(89) xor emCodeBitDelayLine(127);
    when "01101110" =>
        bitCodeBit <= emCodeBitDelayLine(42) xor emInfoBitDelayLine(66) xor emInfoBitDelayLine(105) xor →
            ↪ emInfoBitDelayLine(122) xor emCodeBitDelayLine(127);
    when "01101111" =>
        bitCodeBit <= emInfoBitDelayLine(10) xor emInfoBitDelayLine(33) xor emCodeBitDelayLine(34) xor →
            ↪ emInfoBitDelayLine(88) xor emCodeBitDelayLine(127);
    when "01110000" =>
        bitCodeBit <= emInfoBitDelayLine(2) xor emCodeBitDelayLine(50) xor emInfoBitDelayLine(50) xor →
            ↪ emInfoBitDelayLine(111) xor emCodeBitDelayLine(127);
```

```
when "01110001" =>
    bitCodeBit <= emInfoBitDelayLine(20) xor emInfoBitDelayLine(29) xor emCodeBitDelayLine(43) xor →
        ↪ emInfoBitDelayLine(102) xor emCodeBitDelayLine(127);
when "01110010" =>
    bitCodeBit <= emCodeBitDelayLine(51) xor emInfoBitDelayLine(64) xor emInfoBitDelayLine(78) xor →
        ↪ emInfoBitDelayLine(116) xor emCodeBitDelayLine(127);
when "01110011" =>
    bitCodeBit <= emInfoBitDelayLine(4) xor emInfoBitDelayLine(44) xor emCodeBitDelayLine(54) xor →
        ↪ emInfoBitDelayLine(76) xor emCodeBitDelayLine(127);
when "01110100" =>
    bitCodeBit <= emCodeBitDelayLine(59) xor emInfoBitDelayLine(59) xor emInfoBitDelayLine(88) xor →
        ↪ emInfoBitDelayLine(124) xor emCodeBitDelayLine(127);
when "01110101" =>
    bitCodeBit <= emInfoBitDelayLine(8) xor emCodeBitDelayLine(51) xor emInfoBitDelayLine(61) xor →
        ↪ emInfoBitDelayLine(100) xor emCodeBitDelayLine(127);
when "01110110" =>
    bitCodeBit <= emInfoBitDelayLine(11) xor emCodeBitDelayLine(45) xor emInfoBitDelayLine(55) xor →
        ↪ emInfoBitDelayLine(78) xor emCodeBitDelayLine(127);
when "01110111" =>
    bitCodeBit <= emInfoBitDelayLine(16) xor emCodeBitDelayLine(73) xor emInfoBitDelayLine(74) xor →
        ↪ emInfoBitDelayLine(82) xor emCodeBitDelayLine(127);
when "01111000" =>
    bitCodeBit <= emInfoBitDelayLine(21) xor emCodeBitDelayLine(53) xor emInfoBitDelayLine(59) xor →
        ↪ emInfoBitDelayLine(82) xor emCodeBitDelayLine(127);
when "01111001" =>
    bitCodeBit <= emCodeBitDelayLine(36) xor emInfoBitDelayLine(54) xor emInfoBitDelayLine(119) xor →
        ↪ emInfoBitDelayLine(124) xor emCodeBitDelayLine(127);
when "01111010" =>
    bitCodeBit <= emInfoBitDelayLine(30) xor emInfoBitDelayLine(40) xor emCodeBitDelayLine(77) xor →
        ↪ emInfoBitDelayLine(80) xor emCodeBitDelayLine(127);
when "01111011" =>
    bitCodeBit <= emInfoBitDelayLine(5) xor emCodeBitDelayLine(41) xor emInfoBitDelayLine(53) xor →
        ↪ emInfoBitDelayLine(120) xor emCodeBitDelayLine(127);
when "01111100" =>
    bitCodeBit <= emInfoBitDelayLine(35) xor emCodeBitDelayLine(46) xor emInfoBitDelayLine(65) xor →
        ↪ emInfoBitDelayLine(106) xor emCodeBitDelayLine(127);
when "01111101" =>
    bitCodeBit <= emInfoBitDelayLine(37) xor emInfoBitDelayLine(46) xor emCodeBitDelayLine(70) xor →
        ↪ emInfoBitDelayLine(98) xor emCodeBitDelayLine(127);
when "01111110" =>
    bitCodeBit <= emInfoBitDelayLine(30) xor emCodeBitDelayLine(42) xor emInfoBitDelayLine(46) xor →
        ↪ emInfoBitDelayLine(93) xor emCodeBitDelayLine(127);
when "01111111" =>
    bitCodeBit <= emInfoBitDelayLine(21) xor emInfoBitDelayLine(51) xor emCodeBitDelayLine(77) xor →
        ↪ emInfoBitDelayLine(84) xor emCodeBitDelayLine(127);
when "10000000" =>
    bitCodeBit <= emInfoBitDelayLine(37) xor emInfoBitDelayLine(75) xor emCodeBitDelayLine(79) xor →
        ↪ emInfoBitDelayLine(102) xor emCodeBitDelayLine(127);
when others =>
bitCodeBit <= '0';
end case;
  --End of Auto-generated by VhdlForLdpccEncoder.m see sbates for details.

end process PARITY_GEN;

  -- Take a registered output to help with timing closure and
  -- to be nice to the next block.

ASSIGN_OUTPUT : process ( emCodeBitDelayLine, emInfoBitDelayLine ) is
begin

  bitInfoBitOut <= emInfoBitDelayLine(0);
  bitCodeBitOut <= emCodeBitDelayLine(0);

end process ASSIGN_OUTPUT;


  -- This is the process which updates all the registers
  -- with their new values.

CLOCK_UPDATE : process ( clkClock ) is
begin

 if ( clkClock'event and clkClock='1' ) then

 if ( rstReset = '1' ) then

      emInfoBitDelayLine <= (others=>'0');
  emCodeBitDelayLine <= (others=>'0');
  bitInfoBitIn_Del1  <= '0';
  phPhaseIn_Del1     <= (others=>'0');

    else


      emInfoBitDelayLine <= emInfoBitDelayLine_Next;
  emCodeBitDelayLine <= emCodeBitDelayLine_Next;
  bitInfoBitIn_Del1  <= bitInfoBitIn_Del1_Next;
  phPhaseIn_Del1     <= phPhaseIn_Del1_Next;
```

```
    end if;
    end if;

  end process CLOCK_UPDATE;

end architecture Behavioral;
```

Listing B.5: ChannelEmulator.vhd

```
--------------------------------------------------------------------------------
--
--    Filename: ChannelEmulator.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------------
--
--   Description
--   -----------
--
-- This is a generic channel emulator block which takes (in this case) two binary
-- inputs at a time and emulates a BPSK channel with noise and a AGC and LogLikelihood
-- conversion block. It has a number of parameters which are set by a control
-- input (which also affects other blocks).
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity ChannelEmulator is
    Port ( clkClock         : in LogicBit;
           rstReset         : in LogicBit;
           bitInfoBitIn     : in LogicBit;
           bitCodeBitIn     : in LogicBit;
           chemctrlControl : in std_logic_vector (2 downto 0);
           -- Outputs
           llrInfoLlrOut    : out LogLikeRatio;
           llrCodeLlrOut    : out LogLikeRatio
         );
end entity ChannelEmulator;

architecture Behavioral of ChannelEmulator is

--------------------------------------------------------------------------------
-- COMPONENTS
--------------------------------------------------------------------------------

    -- Components which we instantiate within the main
    -- block.

component LlrSignMagAdd is
    Port ( llrLlrIn1  : in  LogLikeRatio;
           llrLlrIn2  : in  LogLikeRatio;
           llrLlrOut1 : out LogLikeRatio
         );
end component;

component SlvOneCounter is
  port
       ( slvInput  : in std_logic_vector (126 downto 0);
         slvOutput : out std_logic_vector (6 downto 0  )
       );
end component;

--------------------------------------------------------------------------------
-- REGISTERS
--------------------------------------------------------------------------------

    -- Registered values which must have an _Next value
    -- to work well. As per Massana design rules.

signal llrInfoNoiseScaled_Next  : LogLikeRatio;
signal llrInfoNoiseScaled       : LogLikeRatio;

signal llrCodeNoiseScaled_Next  : LogLikeRatio;
```

```
signal llrCodeNoiseScaled        : LogLikeRatio;

signal llrInfoLlrWithNoise_Next  : LogLikeRatio;
signal llrInfoLlrWithNoise       : LogLikeRatio;

signal llrCodeLlrWithNoise_Next  : LogLikeRatio;
signal llrCodeLlrWithNoise       : LogLikeRatio;

signal bitInfoBitIn_Del1_Next    : LogicBit;
signal bitInfoBitIn_Del1         : LogicBit;

signal bitCodeBitIn_Del1_Next    : LogicBit;
signal bitCodeBitIn_Del1         : LogicBit;

signal slvNoiseGenLfsr_Next : std_logic_vector ( 73 downto 0 );
signal slvNoiseGenLfsr      : std_logic_vector ( 73 downto 0 );

signal slvBitsToAdd_Next  : std_logic_vector ( 253 downto 0 );
signal slvBitsToAdd       : std_logic_vector ( 253 downto 0 );

-------------------------------------------------------------------------------
-- COMBINATORIAL
-------------------------------------------------------------------------------

    -- Combinatorial signals that are not directly assigned to a
    -- register.

signal llrInfoLlrNoNoise : LogLikeRatio;  -- The LLR before noise.
signal llrCodeLlrNoNoise : LogLikeRatio;  -- The LLR before noise.

signal llrInfoNoise : LogLikeRatio;
signal llrCodeNoise : LogLikeRatio;

signal slvInfoNoiseTerm : std_logic_vector (6 downto 0);
signal slvCodeNoiseTerm : std_logic_vector (6 downto 0);

signal slvSignalMag         : std_logic_vector (nLlrWidth-1 downto 0);

begin

-------------------------------------------------------------------------------
-- INSTANTIATIONS
-------------------------------------------------------------------------------

InfoNoiseAdder : LlrSignMagAdd
    port map
        ( llrLlrIn1  => llrInfoLlrNoNoise,
          llrLlrIn2  => llrInfoNoiseScaled,
          llrLlrOut1 => llrInfoLlrWithNoise_Next
        );
CodeNoiseAdder : LlrSignMagAdd
    port map
        ( llrLlrIn1  => llrCodeLlrNoNoise,
          llrLlrIn2  => llrCodeNoiseScaled,
          llrLlrOut1 => llrCodeLlrWithNoise_Next
        );
InfoNoiseSum : SlvOneCounter
  port map
      ( slvInput  => slvBitsToAdd(126 downto 0),
        slvOutput => slvInfoNoiseTerm );
CodeNoiseSum : SlvOneCounter
  port map
      ( slvInput  => slvBitsToAdd(253 downto 127),
        slvOutput => slvCodeNoiseTerm );

-------------------------------------------------------------------------------
-- NOISE_LFSR
-------------------------------------------------------------------------------

  -- To generate our noise is a reasonably random way we use a 75 bit LFSR
  -- with primitive polynomial
  --
  -- x^74 + x^10 +x^9 + x + 1
  --
  -- and then take spaced outputs of that and place these in a delay line.
  -- Note this noise is very correlated and uniform. It is for test purposes
  -- only...

  NOISE_LFSR : process ( slvNoiseGenLfsr ) is
  begin

    slvNoiseGenLfsr_Next(0) <= slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(9)
        xor slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(0);
    slvNoiseGenLfsr_Next(73 downto 1) <= slvNoiseGenLfsr(72 downto 0);

  end process NOISE_LFSR;
```

```
-- GENERATE_NOISE
-------------------------------------------------------------------------------

  -- Generate a random term to be subtracted from the information and code
  -- perfect LLRs. Note that this noise should lie in 0 to 2*Mag'length(0)-1
  -- and its power is controlled by ctrlControlVector. Note we can turn the
  -- noise off all together if we wish.

  GENERATE_NOISE : process (slvNoiseGenLfsr, llrInfoNoise, llrCodeNoise, chemctrlControl,
                            slvInfoNoiseTerm, slvCodeNoiseTerm) is
  begin

    -- Our noise is random and semi-normal over N(64,xxx) so if >64 we assume negative.
    -- Note we hould sub 64 from our positive values but inverting the bits is a quick fix.
    -- We also saturate to ensure that we get a good Gaussian approximation.

    llrInfoNoise.Sat <= '0';
    if ( slvInfoNoiseTerm(slvInfoNoiseTerm'left(1)) = '1' ) then
      llrInfoNoise.Sgn <= '1';
      if ( slvInfoNoiseTerm(slvInfoNoiseTerm'left(1)-1 downto 0) > MAX_LLR_NOISE_WIDTH ) then
        llrInfoNoise.Mag <= LLR_MAX_MAG;
      else
        llrInfoNoise.Mag <= slvInfoNoiseTerm(nLlrWidth-1 downto 0);
      end if;
    else
      llrInfoNoise.Sgn <= '0';
      if ( (not slvInfoNoiseTerm(slvInfoNoiseTerm'left(1)-1 downto 0)) > MAX_LLR_NOISE_WIDTH ) then
        llrInfoNoise.Mag <= LLR_MAX_MAG;
      else
        llrInfoNoise.Mag <= not slvInfoNoiseTerm(nLlrWidth-1 downto 0);
      end if;
    end if;

    llrCodeNoise.Sat <= '0';
    if ( slvCodeNoiseTerm(slvCodeNoiseTerm'left(1)) = '1' ) then
      llrCodeNoise.Sgn <= '1';
      if ( slvCodeNoiseTerm(slvCodeNoiseTerm'left(1)-1 downto 0) > MAX_LLR_NOISE_WIDTH ) then
        llrCodeNoise.Mag <= LLR_MAX_MAG;
      else
        llrCodeNoise.Mag <= slvCodeNoiseTerm(nLlrWidth-1 downto 0);
      end if;
    else
      llrCodeNoise.Sgn <= '0';
      if ( (not slvCodeNoiseTerm(slvCodeNoiseTerm'left(1)-1 downto 0)) > MAX_LLR_NOISE_WIDTH ) then
        llrCodeNoise.Mag <= LLR_MAX_MAG;
      else
        llrCodeNoise.Mag <= not slvCodeNoiseTerm(nLlrWidth-1 downto 0);
      end if;
    end if;

    llrInfoNoiseScaled_Next.Sat <= '0';
    llrInfoNoiseScaled_Next.Sgn <= llrInfoNoise.Sgn;

    llrCodeNoiseScaled_Next.Sat <= '0';
    llrCodeNoiseScaled_Next.Sgn <= llrCodeNoise.Sgn;

    -- Based on noise control we either add no noise or our noise. Note
    -- we control the SNR by varying the signal, not the noise. This is
    -- simpler...

    case chemctrlControl is
    when NOISE_OFF =>
      llrInfoNoiseScaled_Next.Mag <= LLR_ZERO_MAG;
      llrCodeNoiseScaled_Next.Mag <= LLR_ZERO_MAG;
    when others =>
      llrInfoNoiseScaled_Next.Mag <= llrInfoNoise.Mag;
      llrCodeNoiseScaled_Next.Mag <= llrCodeNoise.Mag;
    end case;

  end process GENERATE_NOISE;

-------------------------------------------------------------------------------
-- BINARY_TO_LLR
-------------------------------------------------------------------------------

  -- Here we take a binary 1 and map it to half minus maximal and a binary 0 and map it
  -- half positive maximal.

  BINARY_TO_LLR : process (bitInfoBitIn_Del1, bitCodeBitIn_Del1,
                           chemctrlControl, slvSignalMag) is
  begin

    case chemctrlControl is
    when NOISE_OFF =>
      slvSignalMag <= LLR_HALF_MAG;
    when NOISE_ONE =>
      slvSignalMag <= LLR_10_32_MAG;
    when NOISE_TWO =>
      slvSignalMag <= LLR_9_32_MAG;
```

```
      when NOISE_THREE =>
        slvSignalMag <= LLR_QUARTER_MAG;
      when NOISE_FOUR =>
        slvSignalMag <= LLR_7_32_MAG;
      when NOISE_FIVE =>
        slvSignalMag <= LLR_3_16_MAG;
      when NOISE_SIX =>
        slvSignalMag <= LLR_5_32_MAG;
      when NOISE_SEVEN =>
        slvSignalMag <= LLR_EIGHTH_MAG;
      when others =>
        slvSignalMag <= LLR_HALF_MAG;
      end case;

      if ( bitInfoBitIn_Del1 = '1' ) then
        llrInfoLlrNoNoise <= (Sat=>'0',Sgn=>'1',Mag=>slvSignalMag );
      else
        llrInfoLlrNoNoise <= (Sat=>'0',Sgn=>'0',Mag=>slvSignalMag );
      end if;
      if ( bitCodeBitIn_Del1 = '1' ) then
        llrCodeLlrNoNoise <= (Sat=>'0',Sgn=>'1',Mag=>slvSignalMag );
      else
        llrCodeLlrNoNoise <= (Sat=>'0',Sgn=>'0',Mag=>slvSignalMag );
      end if;

    end process BINARY_TO_LLR;

-------------------------------------------------------------------------------
-- ASSIGN_OUTPUT
-------------------------------------------------------------------------------

      -- Take a registered output to help with timing closure and
      -- to be nice to the next block.

    ASSIGN_OUTPUT : process ( llrInfoLlrWithNoise,llrCodeLlrWithNoise ) is
    begin

      llrInfoLlrOut <= llrInfoLlrWithNoise;
      llrCodeLlrOut <= llrCodeLlrWithNoise;

    end process ASSIGN_OUTPUT;

-------------------------------------------------------------------------------
-- DELAY_INPUTS
-------------------------------------------------------------------------------

      -- Register the high-speed inputs.

    DELAY_INPUTS : process ( bitInfoBitIn,bitCodeBitIn ) is
    begin

     bitInfoBitIn_Del1_Next <= bitInfoBitIn;
     bitCodeBitIn_Del1_Next <= bitCodeBitIn;

    end process DELAY_INPUTS;

-------------------------------------------------------------------------------
-- CLOCK_UPDATE
-------------------------------------------------------------------------------

      -- This is the process which updates all the registers
      -- with their new values.

    CLOCK_UPDATE : process ( clkClock ) is
    begin

     if ( clkClock'event and clkClock='1' ) then

     if ( rstReset = '1' ) then

           bitInfoBitIn_Del1 <= '0';
           bitCodeBitIn_Del1 <= '0';

           llrInfoLlrWithNoise <= LOG_LIKE_RATIO_RESET;
           llrCodeLlrWithNoise <= LOG_LIKE_RATIO_RESET;

           llrInfoNoiseScaled <= LOG_LIKE_RATIO_RESET;
           llrCodeNoiseScaled <= LOG_LIKE_RATIO_RESET;


           slvBitsToAdd <= (others=>'0');

              slvNoiseGenLfsr <= (others=>'1');
         else

           bitInfoBitIn_Del1 <= bitInfoBitIn_Del1_Next;
           bitCodeBitIn_Del1 <= bitCodeBitIn_Del1_Next;

           llrInfoLlrWithNoise <= llrInfoLlrWithNoise_Next;
```

```
        llrCodeLlrWithNoise <= llrCodeLlrWithNoise_Next;

        llrInfoNoiseScaled <= llrInfoNoiseScaled_Next;
        llrCodeNoiseScaled <= llrCodeNoiseScaled_Next;

        slvNoiseGenLfsr    <= slvNoiseGenLfsr_Next;
        slvBitsToAdd       <= slvBitsToAdd_Next;


    end if;
    end if;

  end process CLOCK_UPDATE;

-- GEN_BITS_TO_ADD

    -- This is the process which updates all the registers
    -- with their new values.

  GEN_BITS_TO_ADD : process ( slvNoiseGenLfsr ) is
  begin

    --Start of Auto-generated by GenerateVhdlForNoiseGeneration.m see sbates for details.

   slvBitsToAdd_Next(0) <= slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(45) xor slvNoiseGenLfsr(56) xor →
        ↪ slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(16);
   slvBitsToAdd_Next(1) <= slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(29) xor →
        ↪ slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(49);
   slvBitsToAdd_Next(2) <= slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(11) xor →
        ↪ slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(48);
   slvBitsToAdd_Next(3) <= slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(7) xor slvNoiseGenLfsr(38) xor →
        ↪ slvNoiseGenLfsr(28) xor slvNoiseGenLfsr(51);
   slvBitsToAdd_Next(4) <= slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(69) xor slvNoiseGenLfsr(34) xor →
        ↪ slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(15);
   slvBitsToAdd_Next(5) <= slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(60) xor →
        ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(35);
   slvBitsToAdd_Next(6) <= slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(27) xor →
        ↪ slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(45);
   slvBitsToAdd_Next(7) <= slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(56) xor →
        ↪ slvNoiseGenLfsr(2) xor slvNoiseGenLfsr(26);
   slvBitsToAdd_Next(8) <= slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(49) xor →
        ↪ slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(62);
   slvBitsToAdd_Next(9) <= slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(28) xor →
        ↪ slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(23);
   slvBitsToAdd_Next(10) <= slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(60) xor →
        ↪ slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(49);
   slvBitsToAdd_Next(11) <= slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(14) xor slvNoiseGenLfsr(73) xor →
        ↪ slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(1);
   slvBitsToAdd_Next(12) <= slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(61) xor slvNoiseGenLfsr(72) xor →
        ↪ slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(52);
   slvBitsToAdd_Next(13) <= slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(32) xor →
        ↪ slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(51);
   slvBitsToAdd_Next(14) <= slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(9) xor →
        ↪ slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(50);
   slvBitsToAdd_Next(15) <= slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(7) xor →
        ↪ slvNoiseGenLfsr(71) xor slvNoiseGenLfsr(5);
   slvBitsToAdd_Next(16) <= slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(45) xor slvNoiseGenLfsr(9) xor →
        ↪ slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(58);
   slvBitsToAdd_Next(17) <= slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(53) xor →
        ↪ slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(29);
   slvBitsToAdd_Next(18) <= slvNoiseGenLfsr(25) xor slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(55) xor →
        ↪ slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(65);
   slvBitsToAdd_Next(19) <= slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(25) xor slvNoiseGenLfsr(54) xor →
        ↪ slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(56);
   slvBitsToAdd_Next(20) <= slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(55) xor slvNoiseGenLfsr(46) xor →
        ↪ slvNoiseGenLfsr(7) xor slvNoiseGenLfsr(66);
   slvBitsToAdd_Next(21) <= slvNoiseGenLfsr(25) xor slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(52) xor →
        ↪ slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(71);
   slvBitsToAdd_Next(22) <= slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(10) xor →
        ↪ slvNoiseGenLfsr(69) xor slvNoiseGenLfsr(5);
   slvBitsToAdd_Next(23) <= slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(39) xor →
        ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(70);
   slvBitsToAdd_Next(24) <= slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(55) xor slvNoiseGenLfsr(32) xor →
        ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(25);
   slvBitsToAdd_Next(25) <= slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(43) xor →
        ↪ slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(71);
   slvBitsToAdd_Next(26) <= slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(38) xor →
        ↪ slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(68);
   slvBitsToAdd_Next(27) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(4) xor slvNoiseGenLfsr(68) xor →
        ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(44);
   slvBitsToAdd_Next(28) <= slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(25) xor slvNoiseGenLfsr(31) xor →
        ↪ slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(43);
   slvBitsToAdd_Next(29) <= slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(23) xor →
        ↪ slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(12);
   slvBitsToAdd_Next(30) <= slvNoiseGenLfsr(39) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(20) xor →
        ↪ slvNoiseGenLfsr(4) xor slvNoiseGenLfsr(12);
```

```
slvBitsToAdd_Next(31) <= slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(39) xor →
    ↪ slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(45);
slvBitsToAdd_Next(32) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(49) xor →
    ↪ slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(32);
slvBitsToAdd_Next(33) <= slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(13) xor →
    ↪ slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(9);
slvBitsToAdd_Next(34) <= slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(38) xor →
    ↪ slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(5);
slvBitsToAdd_Next(35) <= slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(39) xor slvNoiseGenLfsr(3) xor →
    ↪ slvNoiseGenLfsr(64) xor slvNoiseGenLfsr(38);
slvBitsToAdd_Next(36) <= slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(39) xor slvNoiseGenLfsr(72) xor →
    ↪ slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(16);
slvBitsToAdd_Next(37) <= slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(61) xor →
    ↪ slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(68);
slvBitsToAdd_Next(38) <= slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(65) xor →
    ↪ slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(59);
slvBitsToAdd_Next(39) <= slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(58) xor →
    ↪ slvNoiseGenLfsr(56) xor slvNoiseGenLfsr(11);
slvBitsToAdd_Next(40) <= slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(68) xor slvNoiseGenLfsr(32) xor →
    ↪ slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(35);
slvBitsToAdd_Next(41) <= slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(1) xor →
    ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(12);
slvBitsToAdd_Next(42) <= slvNoiseGenLfsr(25) xor slvNoiseGenLfsr(60) xor slvNoiseGenLfsr(39) xor →
    ↪ slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(31);
slvBitsToAdd_Next(43) <= slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(32) xor →
    ↪ slvNoiseGenLfsr(64) xor slvNoiseGenLfsr(72);
slvBitsToAdd_Next(44) <= slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(30) xor →
    ↪ slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(12);
slvBitsToAdd_Next(45) <= slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(63) xor →
    ↪ slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(38);
slvBitsToAdd_Next(46) <= slvNoiseGenLfsr(60) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(42) xor →
    ↪ slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(67);
slvBitsToAdd_Next(47) <= slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(54) xor →
    ↪ slvNoiseGenLfsr(64) xor slvNoiseGenLfsr(17);
slvBitsToAdd_Next(48) <= slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(14) xor →
    ↪ slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(26);
slvBitsToAdd_Next(49) <= slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(54) xor →
    ↪ slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(57);
slvBitsToAdd_Next(50) <= slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(64) xor →
    ↪ slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(56);
slvBitsToAdd_Next(51) <= slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(72) xor →
    ↪ slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(12);
slvBitsToAdd_Next(52) <= slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(60) xor →
    ↪ slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(52);
slvBitsToAdd_Next(53) <= slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(47) xor →
    ↪ slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(40);
slvBitsToAdd_Next(54) <= slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(18) xor →
    ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(65);
slvBitsToAdd_Next(55) <= slvNoiseGenLfsr(28) xor slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(6) xor →
    ↪ slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(35);
slvBitsToAdd_Next(56) <= slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(42) xor →
    ↪ slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(31);
slvBitsToAdd_Next(57) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(15) xor →
    ↪ slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(58);
slvBitsToAdd_Next(58) <= slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(29) xor →
    ↪ slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(19);
slvBitsToAdd_Next(59) <= slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(0) xor →
    ↪ slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(41);
slvBitsToAdd_Next(60) <= slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(2) xor →
    ↪ slvNoiseGenLfsr(64) xor slvNoiseGenLfsr(26);
slvBitsToAdd_Next(61) <= slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(44) xor →
    ↪ slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(68);
slvBitsToAdd_Next(62) <= slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(63) xor →
    ↪ slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(24);
slvBitsToAdd_Next(63) <= slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(14) xor →
    ↪ slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(68);
slvBitsToAdd_Next(64) <= slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(64) xor →
    ↪ slvNoiseGenLfsr(39) xor slvNoiseGenLfsr(4);
slvBitsToAdd_Next(65) <= slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(3) xor →
    ↪ slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(55);
slvBitsToAdd_Next(66) <= slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(35) xor →
    ↪ slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(0);
slvBitsToAdd_Next(67) <= slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(46) xor →
    ↪ slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(43);
slvBitsToAdd_Next(68) <= slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(31) xor →
    ↪ slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(61);
slvBitsToAdd_Next(69) <= slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(2) xor →
    ↪ slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(30);
slvBitsToAdd_Next(70) <= slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(28) xor →
    ↪ slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(11);
slvBitsToAdd_Next(71) <= slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(54) xor →
    ↪ slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(28);
slvBitsToAdd_Next(72) <= slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(14) xor →
    ↪ slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(43);
slvBitsToAdd_Next(73) <= slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(66) xor →
    ↪ slvNoiseGenLfsr(69) xor slvNoiseGenLfsr(15);
slvBitsToAdd_Next(74) <= slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(44) xor →
    ↪ slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(9);
```

```
slvBitsToAdd_Next(75) <= slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(25) xor →
    ↪ slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(38);
slvBitsToAdd_Next(76) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(68) xor →
    ↪ slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(70);
slvBitsToAdd_Next(77) <= slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(50) xor →
    ↪ slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(19);
slvBitsToAdd_Next(78) <= slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(49) xor →
    ↪ slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(27);
slvBitsToAdd_Next(79) <= slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(62) xor →
    ↪ slvNoiseGenLfsr(28) xor slvNoiseGenLfsr(27);
slvBitsToAdd_Next(80) <= slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(4) xor slvNoiseGenLfsr(40) xor →
    ↪ slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(72);
slvBitsToAdd_Next(81) <= slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(38) xor →
    ↪ slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(17);
slvBitsToAdd_Next(82) <= slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(55) xor →
    ↪ slvNoiseGenLfsr(14) xor slvNoiseGenLfsr(49);
slvBitsToAdd_Next(83) <= slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(44) xor →
    ↪ slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(45);
slvBitsToAdd_Next(84) <= slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(68) xor →
    ↪ slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(42);
slvBitsToAdd_Next(85) <= slvNoiseGenLfsr(63) xor slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(23) xor →
    ↪ slvNoiseGenLfsr(25) xor slvNoiseGenLfsr(34);
slvBitsToAdd_Next(86) <= slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(31) xor →
    ↪ slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(51);
slvBitsToAdd_Next(87) <= slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(25) xor slvNoiseGenLfsr(27) xor →
    ↪ slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(47);
slvBitsToAdd_Next(88) <= slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(32) xor →
    ↪ slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(46);
slvBitsToAdd_Next(89) <= slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(2) xor slvNoiseGenLfsr(16) xor →
    ↪ slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(52);
slvBitsToAdd_Next(90) <= slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(50) xor →
    ↪ slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(10);
slvBitsToAdd_Next(91) <= slvNoiseGenLfsr(7) xor slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(10) xor →
    ↪ slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(59);
slvBitsToAdd_Next(92) <= slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(72) xor →
    ↪ slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(71);
slvBitsToAdd_Next(93) <= slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(34) xor →
    ↪ slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(37);
slvBitsToAdd_Next(94) <= slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(0) xor →
    ↪ slvNoiseGenLfsr(61) xor slvNoiseGenLfsr(47);
slvBitsToAdd_Next(95) <= slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(26) xor →
    ↪ slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(41);
slvBitsToAdd_Next(96) <= slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(23) xor →
    ↪ slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(56);
slvBitsToAdd_Next(97) <= slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(28) xor →
    ↪ slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(59);
slvBitsToAdd_Next(98) <= slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(56) xor →
    ↪ slvNoiseGenLfsr(35) xor slvNoiseGenLfsr(70);
slvBitsToAdd_Next(99) <= slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(51) xor →
    ↪ slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(29);
slvBitsToAdd_Next(100) <= slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(9) xor →
    ↪ slvNoiseGenLfsr(68) xor slvNoiseGenLfsr(54);
slvBitsToAdd_Next(101) <= slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(38) xor →
    ↪ slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(51);
slvBitsToAdd_Next(102) <= slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(66) xor →
    ↪ slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(11);
slvBitsToAdd_Next(103) <= slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(34) xor →
    ↪ slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(60);
slvBitsToAdd_Next(104) <= slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(48) xor →
    ↪ slvNoiseGenLfsr(60) xor slvNoiseGenLfsr(73);
slvBitsToAdd_Next(105) <= slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(42) xor →
    ↪ slvNoiseGenLfsr(35) xor slvNoiseGenLfsr(13);
slvBitsToAdd_Next(106) <= slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(71) xor →
    ↪ slvNoiseGenLfsr(39) xor slvNoiseGenLfsr(16);
slvBitsToAdd_Next(107) <= slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(68) xor →
    ↪ slvNoiseGenLfsr(7) xor slvNoiseGenLfsr(51);
slvBitsToAdd_Next(108) <= slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(49) xor →
    ↪ slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(40);
slvBitsToAdd_Next(109) <= slvNoiseGenLfsr(63) xor slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(44) xor →
    ↪ slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(61);
slvBitsToAdd_Next(110) <= slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(65) xor →
    ↪ slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(71);
slvBitsToAdd_Next(111) <= slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(2) xor slvNoiseGenLfsr(8) xor →
    ↪ slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(7);
slvBitsToAdd_Next(112) <= slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(70) xor →
    ↪ slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(34);
slvBitsToAdd_Next(113) <= slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(0) xor →
    ↪ slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(19);
slvBitsToAdd_Next(114) <= slvNoiseGenLfsr(14) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(32) xor →
    ↪ slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(4);
slvBitsToAdd_Next(115) <= slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(45) xor slvNoiseGenLfsr(24) xor →
    ↪ slvNoiseGenLfsr(39) xor slvNoiseGenLfsr(51);
slvBitsToAdd_Next(116) <= slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(51) xor →
    ↪ slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(2);
slvBitsToAdd_Next(117) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(33) xor →
    ↪ slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(15);
slvBitsToAdd_Next(118) <= slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(56) xor slvNoiseGenLfsr(22) xor →
    ↪ slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(55);
```

```
slvBitsToAdd_Next(119) <= slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(6) xor →
    ↪ slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(71);
slvBitsToAdd_Next(120) <= slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(38) xor →
    ↪ slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(59);
slvBitsToAdd_Next(121) <= slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(42) xor →
    ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(62);
slvBitsToAdd_Next(122) <= slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(9) xor →
    ↪ slvNoiseGenLfsr(14) xor slvNoiseGenLfsr(46);
slvBitsToAdd_Next(123) <= slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(40) xor →
    ↪ slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(3);
slvBitsToAdd_Next(124) <= slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(46) xor →
    ↪ slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(44);
slvBitsToAdd_Next(125) <= slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(24) xor →
    ↪ slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(63);
slvBitsToAdd_Next(126) <= slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(61) xor →
    ↪ slvNoiseGenLfsr(56) xor slvNoiseGenLfsr(26);
slvBitsToAdd_Next(127) <= slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(1) xor →
    ↪ slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(3);
slvBitsToAdd_Next(128) <= slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(30) xor →
    ↪ slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(73);
slvBitsToAdd_Next(129) <= slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(51) xor →
    ↪ slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(40);
slvBitsToAdd_Next(130) <= slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(56) xor →
    ↪ slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(69);
slvBitsToAdd_Next(131) <= slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(68) xor →
    ↪ slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(63);
slvBitsToAdd_Next(132) <= slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(33) xor →
    ↪ slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(71);
slvBitsToAdd_Next(133) <= slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(45) xor →
    ↪ slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(57);
slvBitsToAdd_Next(134) <= slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(45) xor →
    ↪ slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(15);
slvBitsToAdd_Next(135) <= slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(1) xor →
    ↪ slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(72);
slvBitsToAdd_Next(136) <= slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(4) xor →
    ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(15);
slvBitsToAdd_Next(137) <= slvNoiseGenLfsr(28) xor slvNoiseGenLfsr(2) xor slvNoiseGenLfsr(3) xor →
    ↪ slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(24);
slvBitsToAdd_Next(138) <= slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(43) xor →
    ↪ slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(66);
slvBitsToAdd_Next(139) <= slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(45) xor →
    ↪ slvNoiseGenLfsr(2) xor slvNoiseGenLfsr(42);
slvBitsToAdd_Next(140) <= slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(17) xor →
    ↪ slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(59);
slvBitsToAdd_Next(141) <= slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(52) xor →
    ↪ slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(1);
slvBitsToAdd_Next(142) <= slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(65) xor →
    ↪ slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(70);
slvBitsToAdd_Next(143) <= slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(39) xor →
    ↪ slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(41);
slvBitsToAdd_Next(144) <= slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(26) xor →
    ↪ slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(70);
slvBitsToAdd_Next(145) <= slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(4) xor →
    ↪ slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(35);
slvBitsToAdd_Next(146) <= slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(61) xor slvNoiseGenLfsr(11) xor →
    ↪ slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(6);
slvBitsToAdd_Next(147) <= slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(29) xor →
    ↪ slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(58);
slvBitsToAdd_Next(148) <= slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(2) xor slvNoiseGenLfsr(34) xor →
    ↪ slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(67);
slvBitsToAdd_Next(149) <= slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(54) xor →
    ↪ slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(21);
slvBitsToAdd_Next(150) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(56) xor slvNoiseGenLfsr(28) xor →
    ↪ slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(7);
slvBitsToAdd_Next(151) <= slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(45) xor →
    ↪ slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(54);
slvBitsToAdd_Next(152) <= slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(56) xor slvNoiseGenLfsr(41) xor →
    ↪ slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(13);
slvBitsToAdd_Next(153) <= slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(70) xor →
    ↪ slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(29);
slvBitsToAdd_Next(154) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(40) xor →
    ↪ slvNoiseGenLfsr(7) xor slvNoiseGenLfsr(64);
slvBitsToAdd_Next(155) <= slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(33) xor →
    ↪ slvNoiseGenLfsr(35) xor slvNoiseGenLfsr(13);
slvBitsToAdd_Next(156) <= slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(3) xor →
    ↪ slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(57);
slvBitsToAdd_Next(157) <= slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(61) xor →
    ↪ slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(36);
slvBitsToAdd_Next(158) <= slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(3) xor →
    ↪ slvNoiseGenLfsr(45) xor slvNoiseGenLfsr(56);
slvBitsToAdd_Next(159) <= slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(18) xor slvNoiseGenLfsr(12) xor →
    ↪ slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(71);
slvBitsToAdd_Next(160) <= slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(4) xor →
    ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(21);
slvBitsToAdd_Next(161) <= slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(61) xor →
    ↪ slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(28);
slvBitsToAdd_Next(162) <= slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(7) xor slvNoiseGenLfsr(61) xor →
    ↪ slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(54);
```

```
slvBitsToAdd_Next(163) <= slvNoiseGenLfsr(40) xor slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(48) xor →
    ↪ slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(44);
slvBitsToAdd_Next(164) <= slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(28) xor →
    ↪ slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(72);
slvBitsToAdd_Next(165) <= slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(55) xor →
    ↪ slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(0);
slvBitsToAdd_Next(166) <= slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(50) xor →
    ↪ slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(28);
slvBitsToAdd_Next(167) <= slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(4) xor →
    ↪ slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(34);
slvBitsToAdd_Next(168) <= slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(33) xor →
    ↪ slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(26);
slvBitsToAdd_Next(169) <= slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(20) xor →
    ↪ slvNoiseGenLfsr(61) xor slvNoiseGenLfsr(22);
slvBitsToAdd_Next(170) <= slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(69) xor slvNoiseGenLfsr(34) xor →
    ↪ slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(23);
slvBitsToAdd_Next(171) <= slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(48) xor →
    ↪ slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(32);
slvBitsToAdd_Next(172) <= slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(71) xor →
    ↪ slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(55);
slvBitsToAdd_Next(173) <= slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(24) xor →
    ↪ slvNoiseGenLfsr(61) xor slvNoiseGenLfsr(34);
slvBitsToAdd_Next(174) <= slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(45) xor →
    ↪ slvNoiseGenLfsr(71) xor slvNoiseGenLfsr(37);
slvBitsToAdd_Next(175) <= slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(50) xor →
    ↪ slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(21);
slvBitsToAdd_Next(176) <= slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(60) xor slvNoiseGenLfsr(55) xor →
    ↪ slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(7);
slvBitsToAdd_Next(177) <= slvNoiseGenLfsr(55) xor slvNoiseGenLfsr(64) xor slvNoiseGenLfsr(34) xor →
    ↪ slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(58);
slvBitsToAdd_Next(178) <= slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(5) xor →
    ↪ slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(12);
slvBitsToAdd_Next(179) <= slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(35) xor slvNoiseGenLfsr(5) xor →
    ↪ slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(19);
slvBitsToAdd_Next(180) <= slvNoiseGenLfsr(55) xor slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(26) xor →
    ↪ slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(23);
slvBitsToAdd_Next(181) <= slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(69) xor slvNoiseGenLfsr(33) xor →
    ↪ slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(38);
slvBitsToAdd_Next(182) <= slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(6) xor →
    ↪ slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(21);
slvBitsToAdd_Next(183) <= slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(28) xor →
    ↪ slvNoiseGenLfsr(56) xor slvNoiseGenLfsr(29);
slvBitsToAdd_Next(184) <= slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(73) xor →
    ↪ slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(31);
slvBitsToAdd_Next(185) <= slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(46) xor →
    ↪ slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(66);
slvBitsToAdd_Next(186) <= slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(59) xor →
    ↪ slvNoiseGenLfsr(71) xor slvNoiseGenLfsr(1);
slvBitsToAdd_Next(187) <= slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(56) xor →
    ↪ slvNoiseGenLfsr(69) xor slvNoiseGenLfsr(61);
slvBitsToAdd_Next(188) <= slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(39) xor →
    ↪ slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(69);
slvBitsToAdd_Next(189) <= slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(62) xor →
    ↪ slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(40);
slvBitsToAdd_Next(190) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(15) xor →
    ↪ slvNoiseGenLfsr(35) xor slvNoiseGenLfsr(6);
slvBitsToAdd_Next(191) <= slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(64) xor slvNoiseGenLfsr(31) xor →
    ↪ slvNoiseGenLfsr(8) xor slvNoiseGenLfsr(44);
slvBitsToAdd_Next(192) <= slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(34) xor →
    ↪ slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(17);
slvBitsToAdd_Next(193) <= slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(12) xor →
    ↪ slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(63);
slvBitsToAdd_Next(194) <= slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(70) xor →
    ↪ slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(48);
slvBitsToAdd_Next(195) <= slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(64) xor →
    ↪ slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(70);
slvBitsToAdd_Next(196) <= slvNoiseGenLfsr(60) xor slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(53) xor →
    ↪ slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(70);
slvBitsToAdd_Next(197) <= slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(60) xor slvNoiseGenLfsr(29) xor →
    ↪ slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(35);
slvBitsToAdd_Next(198) <= slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(16) xor →
    ↪ slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(7);
slvBitsToAdd_Next(199) <= slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(32) xor →
    ↪ slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(34);
slvBitsToAdd_Next(200) <= slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(40) xor →
    ↪ slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(20);
slvBitsToAdd_Next(201) <= slvNoiseGenLfsr(60) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(61) xor →
    ↪ slvNoiseGenLfsr(71) xor slvNoiseGenLfsr(4);
slvBitsToAdd_Next(202) <= slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(55) xor slvNoiseGenLfsr(44) xor →
    ↪ slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(18);
slvBitsToAdd_Next(203) <= slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(45) xor slvNoiseGenLfsr(72) xor →
    ↪ slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(60);
slvBitsToAdd_Next(204) <= slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(19) xor →
    ↪ slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(55);
slvBitsToAdd_Next(205) <= slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(29) xor slvNoiseGenLfsr(8) xor →
    ↪ slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(40);
slvBitsToAdd_Next(206) <= slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(30) xor →
    ↪ slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(16);
```

```
slvBitsToAdd_Next(207) <= slvNoiseGenLfsr(4) xor slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(19) xor →
    ↪ slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(42);
slvBitsToAdd_Next(208) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(45) xor slvNoiseGenLfsr(55) xor →
    ↪ slvNoiseGenLfsr(22) xor slvNoiseGenLfsr(39);
slvBitsToAdd_Next(209) <= slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(14) xor slvNoiseGenLfsr(36) xor →
    ↪ slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(48);
slvBitsToAdd_Next(210) <= slvNoiseGenLfsr(46) xor slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(52) xor →
    ↪ slvNoiseGenLfsr(67) xor slvNoiseGenLfsr(65);
slvBitsToAdd_Next(211) <= slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(57) xor →
    ↪ slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(18);
slvBitsToAdd_Next(212) <= slvNoiseGenLfsr(39) xor slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(71) xor →
    ↪ slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(73);
slvBitsToAdd_Next(213) <= slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(41) xor slvNoiseGenLfsr(73) xor →
    ↪ slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(67);
slvBitsToAdd_Next(214) <= slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(22) xor →
    ↪ slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(69);
slvBitsToAdd_Next(215) <= slvNoiseGenLfsr(71) xor slvNoiseGenLfsr(35) xor slvNoiseGenLfsr(38) xor →
    ↪ slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(36);
slvBitsToAdd_Next(216) <= slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(68) xor slvNoiseGenLfsr(13) xor →
    ↪ slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(31);
slvBitsToAdd_Next(217) <= slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(33) xor →
    ↪ slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(23);
slvBitsToAdd_Next(218) <= slvNoiseGenLfsr(20) xor slvNoiseGenLfsr(39) xor slvNoiseGenLfsr(58) xor →
    ↪ slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(71);
slvBitsToAdd_Next(219) <= slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(73) xor slvNoiseGenLfsr(55) xor →
    ↪ slvNoiseGenLfsr(11) xor slvNoiseGenLfsr(41);
slvBitsToAdd_Next(220) <= slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(1) xor slvNoiseGenLfsr(17) xor →
    ↪ slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(71);
slvBitsToAdd_Next(221) <= slvNoiseGenLfsr(6) xor slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(33) xor →
    ↪ slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(32);
slvBitsToAdd_Next(222) <= slvNoiseGenLfsr(35) xor slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(51) xor →
    ↪ slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(5);
slvBitsToAdd_Next(223) <= slvNoiseGenLfsr(50) xor slvNoiseGenLfsr(60) xor slvNoiseGenLfsr(0) xor →
    ↪ slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(44);
slvBitsToAdd_Next(224) <= slvNoiseGenLfsr(4) xor slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(39) xor →
    ↪ slvNoiseGenLfsr(68) xor slvNoiseGenLfsr(28);
slvBitsToAdd_Next(225) <= slvNoiseGenLfsr(12) xor slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(30) xor →
    ↪ slvNoiseGenLfsr(56) xor slvNoiseGenLfsr(21);
slvBitsToAdd_Next(226) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(32) xor →
    ↪ slvNoiseGenLfsr(7) xor slvNoiseGenLfsr(39);
slvBitsToAdd_Next(227) <= slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(21) xor →
    ↪ slvNoiseGenLfsr(62) xor slvNoiseGenLfsr(66);
slvBitsToAdd_Next(228) <= slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(33) xor slvNoiseGenLfsr(72) xor →
    ↪ slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(63);
slvBitsToAdd_Next(229) <= slvNoiseGenLfsr(24) xor slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(58) xor →
    ↪ slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(5);
slvBitsToAdd_Next(230) <= slvNoiseGenLfsr(4) xor slvNoiseGenLfsr(3) xor slvNoiseGenLfsr(40) xor →
    ↪ slvNoiseGenLfsr(14) xor slvNoiseGenLfsr(72);
slvBitsToAdd_Next(231) <= slvNoiseGenLfsr(55) xor slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(62) xor →
    ↪ slvNoiseGenLfsr(36) xor slvNoiseGenLfsr(1);
slvBitsToAdd_Next(232) <= slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(31) xor slvNoiseGenLfsr(33) xor →
    ↪ slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(13);
slvBitsToAdd_Next(233) <= slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(43) xor slvNoiseGenLfsr(32) xor →
    ↪ slvNoiseGenLfsr(70) xor slvNoiseGenLfsr(58);
slvBitsToAdd_Next(234) <= slvNoiseGenLfsr(61) xor slvNoiseGenLfsr(64) xor slvNoiseGenLfsr(6) xor →
    ↪ slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(0);
slvBitsToAdd_Next(235) <= slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(2) xor slvNoiseGenLfsr(14) xor →
    ↪ slvNoiseGenLfsr(52) xor slvNoiseGenLfsr(58);
slvBitsToAdd_Next(236) <= slvNoiseGenLfsr(42) xor slvNoiseGenLfsr(69) xor slvNoiseGenLfsr(57) xor →
    ↪ slvNoiseGenLfsr(66) xor slvNoiseGenLfsr(72);
slvBitsToAdd_Next(237) <= slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(22) xor →
    ↪ slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(28);
slvBitsToAdd_Next(238) <= slvNoiseGenLfsr(27) xor slvNoiseGenLfsr(57) xor slvNoiseGenLfsr(18) xor →
    ↪ slvNoiseGenLfsr(21) xor slvNoiseGenLfsr(28);
slvBitsToAdd_Next(239) <= slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(60) xor →
    ↪ slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(34);
slvBitsToAdd_Next(240) <= slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(0) xor slvNoiseGenLfsr(6) xor →
    ↪ slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(31);
slvBitsToAdd_Next(241) <= slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(49) xor slvNoiseGenLfsr(6) xor →
    ↪ slvNoiseGenLfsr(5) xor slvNoiseGenLfsr(33);
slvBitsToAdd_Next(242) <= slvNoiseGenLfsr(32) xor slvNoiseGenLfsr(54) xor slvNoiseGenLfsr(70) xor →
    ↪ slvNoiseGenLfsr(30) xor slvNoiseGenLfsr(11);
slvBitsToAdd_Next(243) <= slvNoiseGenLfsr(14) xor slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(34) xor →
    ↪ slvNoiseGenLfsr(37) xor slvNoiseGenLfsr(69);
slvBitsToAdd_Next(244) <= slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(28) xor →
    ↪ slvNoiseGenLfsr(10) xor slvNoiseGenLfsr(39);
slvBitsToAdd_Next(245) <= slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(69) xor slvNoiseGenLfsr(65) xor →
    ↪ slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(23);
slvBitsToAdd_Next(246) <= slvNoiseGenLfsr(15) xor slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(13) xor →
    ↪ slvNoiseGenLfsr(58) xor slvNoiseGenLfsr(66);
slvBitsToAdd_Next(247) <= slvNoiseGenLfsr(17) xor slvNoiseGenLfsr(28) xor slvNoiseGenLfsr(51) xor →
    ↪ slvNoiseGenLfsr(53) xor slvNoiseGenLfsr(11);
slvBitsToAdd_Next(248) <= slvNoiseGenLfsr(48) xor slvNoiseGenLfsr(13) xor slvNoiseGenLfsr(3) xor →
    ↪ slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(57);
slvBitsToAdd_Next(249) <= slvNoiseGenLfsr(9) xor slvNoiseGenLfsr(25) xor slvNoiseGenLfsr(65) xor →
    ↪ slvNoiseGenLfsr(23) xor slvNoiseGenLfsr(16);
slvBitsToAdd_Next(250) <= slvNoiseGenLfsr(47) xor slvNoiseGenLfsr(56) xor slvNoiseGenLfsr(59) xor →
    ↪ slvNoiseGenLfsr(72) xor slvNoiseGenLfsr(37);
```

```
    slvBitsToAdd_Next(251) <= slvNoiseGenLfsr(59) xor slvNoiseGenLfsr(34) xor slvNoiseGenLfsr(18) xor →
        ↪ slvNoiseGenLfsr(44) xor slvNoiseGenLfsr(69);
    slvBitsToAdd_Next(252) <= slvNoiseGenLfsr(51) xor slvNoiseGenLfsr(19) xor slvNoiseGenLfsr(50) xor →
        ↪ slvNoiseGenLfsr(26) xor slvNoiseGenLfsr(35);
    slvBitsToAdd_Next(253) <= slvNoiseGenLfsr(38) xor slvNoiseGenLfsr(65) xor slvNoiseGenLfsr(53) xor →
        ↪ slvNoiseGenLfsr(16) xor slvNoiseGenLfsr(54);

  end process GEN_BITS_TO_ADD;

end architecture Behavioral;
```

Listing B.6: InfoBitGenerator.vhd

```
--------------------------------------------------------------------------------
--
--    Filename: InfoBitGenerator.vhd
--
--    Created:  06 July 2004
--    Modified: 01 September 2004
--    Version:  $
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------------------
--
--   Description
--   -----------
--
-- This is a standard 32-bit Linear Feedback Shift Register (LFSR), that
-- takes two binary inputs, a 32-bit vector and outputs one bit.
-- This LFSR generates pseudo-random numbers ie. 2^32-1 = 4,294,967,295
-- numbers. Also includes other test mode sequences.
--
-- INPUTS:
--
-- OUTPUTS:
--
-- References: This code is adapted from the original. The original resides on
-- the EE552 application notes:
-- http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/
-- Drivers_Ed/lfsr.html
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity InfoBitGenerator is

  generic (
    nNumBits: integer := 32
    );

  port (
      -- inputs
      clkClock      : in  std_logic;          -- clock
      slLoad        : in  std_logic;          -- active high load
      rstReset      : in  std_logic;          -- active high reset
      slvSeed       : in  std_logic_vector(nNumBits-1 downto 0);  -- initial parallel seed value
      slvTestPattern: in  std_logic_vector(2 downto 0);  -- type of test pattern
      -- outputs
      phPhaseOut  : out Phase;
      slSerialOut : out std_logic);        -- serial output (rightmost register)
end InfoBitGenerator;

architecture pseudo_sequence of InfoBitGenerator is

  signal slvLfsrReg: std_logic_vector(nNumBits-1 downto 0);
  signal slvLfsrReg_Next : std_logic_vector(nNumBits-1 downto 0);

  signal slAlternatingBit : std_logic;          -- value for the alternate test sequence
  signal slAlternatingBit_Next : std_logic;

  signal slInfoBitDelay1 : std_logic;
  signal slInfoBitDelay1_Next : std_logic;

  signal slInfoBitDelay2 : std_logic;
  signal slInfoBitDelay2_Next : std_logic;

  signal slInfoBitDelay3 : std_logic;
  signal slInfoBitDelay3_Next : std_logic;
```

```
  signal slInfoBitDelay4 : std_logic;
  signal slInfoBitDelay4_Next : std_logic;

  signal phPhase      : Phase;
  signal phPhase_Next : Phase;


-- Tap configuration: ones depict a feedback connection (occur at 1,5,6,31)
  signal slvTapConfig: std_logic_vector (nNumBits-1 downto 0);


begin  -- pseudo_sequence
-----------------------------------------------------------------------
-- ASSIGN_OUTPUT
-- This process only assigns the output wire with the correct test pattern
-- result. A TestPattern is required to select among the different modes.
-----------------------------------------------------------------------
    ASSIGN_OUTPUT : process( slAlternatingBit , slvLfsrReg , slInfoBitDelay4 ,
                             slvTestPattern , phPhase )
    begin
        case slvTestPattern is
            when "000" =>
                slSerialOut <= slvLfsrReg(nNumBits-1);
            when "001" =>
                slSerialOut <= slAlternatingBit;
            when "010" =>
                slSerialOut <= slInfoBitDelay4;
            when others =>
                slSerialOut <= '0';
        end case;
        phPhaseOut <= phPhase;

    end process ASSIGN_OUTPUT;

-----------------------------------------------------------------------
-- DATA_LFSR
-----------------------------------------------------------------------

  -- To generate our data is a reasonably random way we use a 32 bit LFSR
  -- with primitive polynomial
  --
  -- x^32 + x^7 +x^6 + x^2 + 1
  --
  -- and then take spaced outputs of that and place these in a delay line.
  -- Note the data is very correlated and uniform. It is for test purposes
  -- only...

  DATA_LFSR : process ( slvLfsrReg , slLoad , slvSeed ) is
  begin

        if ( slLoad = '1' ) then
         slvLfsrReg_Next <= slvSeed;
        else
      slvLfsrReg_Next(0) <= slvLfsrReg(31) xor slvLfsrReg(6)
        xor slvLfsrReg(5) xor slvLfsrReg(1);
      slvLfsrReg_Next(31 downto 1) <= slvLfsrReg(30 downto 0);
    end if;

  end process DATA_LFSR;

-----------------------------------------------------------------------
-- TEST_SEQUENCE
-- This process outputs the values in the registers, initially set, and
-- then outputs zeroes forever.
-----------------------------------------------------------------------

    TEST_SEQUENCE: process ( slInfoBitDelay1 , slInfoBitDelay2 , slInfoBitDelay3 ,
                             slAlternatingBit )
    begin  -- process TEST_SEQUENCE

      slInfoBitDelay1_Next <= '0';
      slInfoBitDelay2_Next <= slInfoBitDelay1;
      slInfoBitDelay3_Next <= slInfoBitDelay2;
      slInfoBitDelay4_Next <= slInfoBitDelay3;

      slAlternatingBit_Next <= slAlternatingBit xor '1';

    end process TEST_SEQUENCE;

-----------------------------------------------------------------------
-- CLOCK_UPDATE
-----------------------------------------------------------------------

   PHASE_COUNTER : process ( phPhase ) is
   begin
    if ( phPhase = PHASE_MAXIMUM ) then
        phPhase_Next <= (others=>'0');
    else
        phPhase_Next <= phPhase + "00000001";
```

```
    end if;
   end process PHASE_COUNTER;

-------------------------------------------------------------------------------
-- CLOCK_UPDATE
-------------------------------------------------------------------------------

   CLOCK_UPDATE : process ( clkClock ) is
   begin

    if ( clkClock'event and clkClock='1' ) then

          if ( rstReset = '1' ) then
              slvTapConfig    <= "10000000000000000000000001100010";
              slAlternatingBit<= '1';
              slvLfsrReg      <= (others => '1');
              slInfoBitDelay1 <= '1';
              slInfoBitDelay2 <= '1';
              slInfoBitDelay3 <= '1';
              slInfoBitDelay4 <= '1';
              phPhase         <= (others=>'0');
          else
              slAlternatingBit        <= slAlternatingBit_Next;
              slvLfsrReg              <= slvLfsrReg_Next;
              slInfoBitDelay1         <= slInfoBitDelay1_Next;
              slInfoBitDelay2         <= slInfoBitDelay2_Next;
              slInfoBitDelay3         <= slInfoBitDelay3_Next;
              slInfoBitDelay4         <= slInfoBitDelay4_Next;
              phPhase                 <= phPhase_Next;
          end if;
    end if;

   end process CLOCK_UPDATE;

end pseudo_sequence;
```

Listing B.7: SlvOneCounter.vhd

```
-------------------------------------------------------------------------------
--
--    Filename: SlvOneCounter.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ----------------------------------------------------------------------------
--
--   Description
--   -----------
--
-- This is a generic channel emulator block which takes (in this case) two binary
-- inputs at a time and emulates a BPSK channel with noise and a AGC and LogLiklihood
-- conversion block. It has a number of parameters which are set by a control
-- input (which also affects other blocks).
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity SlvOneCounter is
  port
       ( slvInput  : in std_logic_vector (126 downto 0);
         slvOutput : out std_logic_vector (6 downto 0  )
       );
end entity;

architecture Behavioral of SlvOneCounter is

-------------------------------------------------------------------------------
-- COMPONENTS
-------------------------------------------------------------------------------

     -- Components which we instantiate within the main
     -- block.

signal slvInputSignExt : NoiseArray;
```

```
begin

-------------------------------------------------------------------------------
-- SIGN_EXTEND_INPUT
-------------------------------------------------------------------------------

   GENERATE_NOISE : process (slvInput) is
   begin

   for nThisReg in 0 to 126 loop
     slvInputSignExt(nThisReg)(0) <= slvInput(nThisReg);
     slvInputSignExt(nThisReg)(6 downto 1) <= "000000";
   end loop;

   end process GENERATE_NOISE;

-------------------------------------------------------------------------------
-- ADD_IT_UP
-------------------------------------------------------------------------------

   ADD_IT_UP : process (slvInputSignExt) is
   begin

    slvOutput <=
         slvInputSignExt(0) +      slvInputSignExt(1) +      slvInputSignExt(2) +      slvInputSignExt(3) + →
              ↪      slvInputSignExt(4) +      slvInputSignExt(5) +      slvInputSignExt(6) +      →
              ↪ slvInputSignExt(7) +      slvInputSignExt(8) +      slvInputSignExt(9) +      →
              ↪ slvInputSignExt(10) +      slvInputSignExt(11) +      slvInputSignExt(12) +      →
              ↪ slvInputSignExt(13) +      slvInputSignExt(14) +      slvInputSignExt(15) +      →
              ↪ slvInputSignExt(16) +      slvInputSignExt(17) +      slvInputSignExt(18) +      →
              ↪ slvInputSignExt(19) +      slvInputSignExt(20) +      slvInputSignExt(21) +      →
              ↪ slvInputSignExt(22) +      slvInputSignExt(23) +      slvInputSignExt(24) +      →
              ↪ slvInputSignExt(25) +      slvInputSignExt(26) +      slvInputSignExt(27) +      →
              ↪ slvInputSignExt(28) +      slvInputSignExt(29) +      slvInputSignExt(30) +      →
              ↪ slvInputSignExt(31) +      slvInputSignExt(32) +      slvInputSignExt(33) +      →
              ↪ slvInputSignExt(34) +      slvInputSignExt(35) +      slvInputSignExt(36) +      →
              ↪ slvInputSignExt(37) +      slvInputSignExt(38) +      slvInputSignExt(39) +      →
              ↪ slvInputSignExt(40) +      slvInputSignExt(41) +      slvInputSignExt(42) +      →
              ↪ slvInputSignExt(43) +      slvInputSignExt(44) +      slvInputSignExt(45) +      →
              ↪ slvInputSignExt(46) +      slvInputSignExt(47) +      slvInputSignExt(48) +      →
              ↪ slvInputSignExt(49) +      slvInputSignExt(50) +      slvInputSignExt(51) +      →
              ↪ slvInputSignExt(52) +      slvInputSignExt(53) +      slvInputSignExt(54) +      →
              ↪ slvInputSignExt(55) +      slvInputSignExt(56) +      slvInputSignExt(57) +      →
              ↪ slvInputSignExt(58) +      slvInputSignExt(59) +      slvInputSignExt(60) +      →
              ↪ slvInputSignExt(61) +      slvInputSignExt(62) +      slvInputSignExt(63) +      →
              ↪ slvInputSignExt(64) +      slvInputSignExt(65) +      slvInputSignExt(66) +      →
              ↪ slvInputSignExt(67) +      slvInputSignExt(68) +      slvInputSignExt(69) +      →
              ↪ slvInputSignExt(70) +      slvInputSignExt(71) +      slvInputSignExt(72) +      →
              ↪ slvInputSignExt(73) +      slvInputSignExt(74) +      slvInputSignExt(75) +      →
              ↪ slvInputSignExt(76) +      slvInputSignExt(77) +      slvInputSignExt(78) +      →
              ↪ slvInputSignExt(79) +      slvInputSignExt(80) +      slvInputSignExt(81) +      →
              ↪ slvInputSignExt(82) +      slvInputSignExt(83) +      slvInputSignExt(84) +      →
              ↪ slvInputSignExt(85) +      slvInputSignExt(86) +      slvInputSignExt(87) +      →
              ↪ slvInputSignExt(88) +      slvInputSignExt(89) +      slvInputSignExt(90) +      →
              ↪ slvInputSignExt(91) +      slvInputSignExt(92) +      slvInputSignExt(93) +      →
              ↪ slvInputSignExt(94) +      slvInputSignExt(95) +      slvInputSignExt(96) +      →
              ↪ slvInputSignExt(97) +      slvInputSignExt(98) +      slvInputSignExt(99) +      →
              ↪ slvInputSignExt(100) +      slvInputSignExt(101) +      slvInputSignExt(102) +      →
              ↪ slvInputSignExt(103) +      slvInputSignExt(104) +      slvInputSignExt(105) +      →
              ↪ slvInputSignExt(106) +      slvInputSignExt(107) +      slvInputSignExt(108) +      →
              ↪ slvInputSignExt(109) +      slvInputSignExt(110) +      slvInputSignExt(111) +      →
              ↪ slvInputSignExt(112) +      slvInputSignExt(113) +      slvInputSignExt(114) +      →
              ↪ slvInputSignExt(115) +      slvInputSignExt(116) +      slvInputSignExt(117) +      →
              ↪ slvInputSignExt(118) +      slvInputSignExt(119) +      slvInputSignExt(120) +      →
              ↪ slvInputSignExt(121) +      slvInputSignExt(122) +      slvInputSignExt(123) +      →
              ↪ slvInputSignExt(124) +      slvInputSignExt(125) +      slvInputSignExt(126);

end process ADD_IT_UP;

end Behavioral;
```

Listing B.8: Icfaalp2OutputInterface.vhd

```
-------------------------------------------------------------------------------
--
--    Filename: Icfaalp2OutputInterface.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ----------------------------------------------------------------------------
--
```

```
--    Description
--    -----------
--
--    This is the VHDL file for the ICFAALP2 output interface which is to be
--    fabricated by TSMC 0.18u.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

  -- Caveat Empore!! The package below has the same name as for other projects but
  -- it should be loaded from the LCFAALP1 directory to ensure correct word lengths
  -- etc. THIS WILL BE DONE FOR YOU IF YOU USE THE Lcfaalp1Script.do SCRIPT IN THE
  -- MODELSIM DIRECTORY!!!!

use work.LdpccPackageForIcfaalp2.ALL;

entity OutputInterface is
    Port ( -- Inputs from Pins
           slvControlFp             : in std_logic_vector ( NUM_CONTROL_BITS-1     downto 0 );
           -- Common Inputs
           clkClock                 : in LogicBit;
           rstReset                 : in LogicBit;
           -- Inputs from Encoder
           bitInfoBitInEnc   : in LogicBit;
           bitCodeBitInEnc   : in LogicBit;
           -- Inputs from Input Interface;
           llrduInfoLlrDelayUnitInIpIf  : in LogLikeRatioDelayUnit;
           llrduCodeLlrDelayUnitInIpIf  : in LogLikeRatioDelayUnit;
           phPhaseInIpIf                : in Phase;
           phaPhaseOffsetInIpIf         : in PhaseArray;
           -- Inputs from Processors
           llrduaInfoLlrDelayUnitInPro  : in LogLikeRatioDelayUnitArray;
           llrduaCodeLlrDelayUnitInPro  : in LogLikeRatioDelayUnitArray;
           -- Inputs from Sign Extractor
           lbInfoBitInSgnEx             : in LogicBit;
           lbCodeBitInSgnEx             : in LogicBit;
           -- Inputs from Error Counter
           slvBlockCountLog2InErr       : in std_logic_vector (7 downto 0);
           -- Outputs to Processors
           phPhaseOutPro                : out Phase;

           phaPhaseOffsetOutPro         : out PhaseArray;
           llrduaInfoLlrDelayUnitOutPro : out LogLikeRatioDelayUnitArray;
           llrduaCodeLlrDelayUnitOutPro : out LogLikeRatioDelayUnitArray;

           -- Outputs to Sign Extractors
           llrduInfoLleDelayUnitOutSignEx : out LogLikeRatioDelayUnit;
           llrduCodeLleDelayUnitOutSignEx : out LogLikeRatioDelayUnit;
           -- Outputs to Pins
           slvHighSpeedOutputsTp    : out std_logic_vector ( NUM_HIGH_SPEED_OPS-1 downto 0 )
           );
end entity OutputInterface;

architecture Behavioral of OutputInterface is

signal slvHighSpeedOutputsTpInt   : std_logic_vector ( 2*(nLlrWidth+2)-1 downto 0 );

signal slvHighSpeedOutputsTp_Reg : std_logic_vector ( NUM_HIGH_SPEED_OPS-1 downto 0 );
signal slvHighSpeedOutputsTp_Reg_Next : std_logic_vector ( NUM_HIGH_SPEED_OPS-1 downto 0 );


begin

  -- This entity is simply a big mux which is driven by the control signal
  -- slvControlFp. Depending on its value certain inputs get driven to
  -- certain outputs.

--------------------------------------------------------------------------------
-- DIRECT_ASSIGNMENTS
--------------------------------------------------------------------------------

  -- This process always assigns certain signals to certain outputs regardless of the
  -- input control signal.

  DIRECT_ASSIGNMENTS : process ( phPhaseInIpIf, bitInfoBitInEnc, bitCodeBitInEnc,
                lbInfoBitInSgnEx, lbCodeBitInSgnEx, slvControlFp, slvBlockCountLog2InErr,
                slvHighSpeedOutputsTpInt, slvHighSpeedOutputsTp_Reg ) is
  begin

    --slvHighSpeedOutputsTp(slvHighSpeedOutputsTp'left(1) downto 15) <= (others=>'0');
    slvHighSpeedOutputsTp_Reg_Next(2*(nLlrWidth+2)+3) <= lbInfoBitInSgnEx;
    slvHighSpeedOutputsTp_Reg_Next(2*(nLlrWidth+2)+2) <= lbCodeBitInSgnEx;
    slvHighSpeedOutputsTp_Reg_Next(2*(nLlrWidth+2)+1) <= bitInfoBitInEnc;
    slvHighSpeedOutputsTp_Reg_Next(2*(nLlrWidth+2))   <= bitCodeBitInEnc;
    phPhaseOutPro                                     <= phPhaseInIpIf;
```

```
    slvHighSpeedOutputsTp<= slvHighSpeedOutputsTp_Reg;

    -- Some of the outputs of the high spee dbus can vary depending
    -- on the input control vector. In fact it works like this.
    -- XXX.XX11 implies output control vector on output (for test)
    -- 1XXXXX0X implies output BER estimate
    -- 0XXX.0X  implies LLRs.

    if ( slvControlFp(1 downto 0) = "11") then
      slvHighSpeedOutputsTp_Reg_Next <= slvControlFp ( NUM_HIGH_SPEED_OPS-1 downto 0 );
    else
    if ( slvControlFp(17) = '1' ) then
       slvHighSpeedOutputsTp_Reg_Next(7 downto 0)                    <= slvBlockCountLog2InErr;
       slvHighSpeedOutputsTp_Reg_Next(2*(nLlrWidth+2)-1 downto 8) <= slvHighSpeedOutputsTpInt(2*(nLlrWidth →
            ↪ +2)-1 downto 8);
    else
       slvHighSpeedOutputsTp_Reg_Next(2*(nLlrWidth+2)-1 downto 0) <= slvHighSpeedOutputsTpInt(2*(nLlrWidth →
            ↪ +2)-1 downto 0);
    end if;
    end if;


  end process DIRECT_ASSIGNMENTS;

-------------------------------------------------------------------------------
-- CLOCK_UPDATE
-------------------------------------------------------------------------------

    -- This is the process which updates all the registers
    -- with their new values.

  CLOCK_UPDATE : process ( clkClock ) is
  begin

     if ( clkClock'event and clkClock='1' ) then

        if ( rstReset = '1' ) then

          slvHighSpeedOutputsTp_Reg <= (others=>'0');

        else

          slvHighSpeedOutputsTp_Reg <= slvHighSpeedOutputsTp_Reg_Next;

        end if;

     end if;

   end process CLOCK_UPDATE;


-------------------------------------------------------------------------------
-- PROCESSOR_INPUTS
-------------------------------------------------------------------------------

  -- We can take the LLRs from the InputInterface and place them into any
  -- one of the processors. We then take the outputs to the inputs and loop
  -- around to the beginning. This allows us to allocate any processor as
  -- the 1st one.

   PROCESSOR_INPUTS : process ( phaPhaseOffsetInIpIf, llrduInfoLlrDelayUnitInIpIf,
                                llrduCodeLlrDelayUnitInIpIf, llrduaInfoLlrDelayUnitInPro,
                                llrduaCodeLlrDelayUnitInPro, slvControlFp ) is
   begin


case slvControlFp( 7 downto 4 ) is
when "0000" =>
phaPhaseOffsetOutPro <= phaPhaseOffsetInIpIf;
llrduaInfoLlrDelayUnitOutPro(0)                          <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 1) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
      ↪ downto 0);
llrduaCodeLlrDelayUnitOutPro(0)                          <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 1) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
      ↪ downto 0);
when "0001" =>
phaPhaseOffsetOutPro(nNumProcessors-1 downto 1) <= phaPhaseOffsetInIpIf(8 downto 0);
phaPhaseOffsetOutPro(1-1 downto 0)              <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 9);
llrduaInfoLlrDelayUnitOutPro(1) <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 2) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
      ↪ downto 1);
llrduaInfoLlrDelayUnitOutPro(0) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(1) <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 2) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
      ↪ downto 1);
llrduaCodeLlrDelayUnitOutPro(0) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
when "0010" =>
phaPhaseOffsetOutPro(nNumProcessors-1 downto 2) <= phaPhaseOffsetInIpIf(7 downto 0);
```

```
phaPhaseOffsetOutPro(2-1 downto 0)              <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 8);
llrduaInfoLlrDelayUnitOutPro(2) <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 3) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 2);
llrduaInfoLlrDelayUnitOutPro(1 downto 1) <= llrduaInfoLlrDelayUnitInPro(0 downto 0);
llrduaInfoLlrDelayUnitOutPro(0) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(2) <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 3) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 2);
llrduaCodeLlrDelayUnitOutPro(1 downto 1) <= llrduaCodeLlrDelayUnitInPro(0 downto 0);
llrduaCodeLlrDelayUnitOutPro(0) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
when "0011" =>
phaPhaseOffsetOutPro(nNumProcessors-1 downto 3) <= phaPhaseOffsetInIpIf(6 downto 0);
phaPhaseOffsetOutPro(3-1 downto 0)              <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 7);
llrduaInfoLlrDelayUnitOutPro(3) <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 4) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 3);
llrduaInfoLlrDelayUnitOutPro(2 downto 1) <= llrduaInfoLlrDelayUnitInPro(1 downto 0);
llrduaInfoLlrDelayUnitOutPro(0) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(3) <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 4) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 3);
llrduaCodeLlrDelayUnitOutPro(2 downto 1) <= llrduaCodeLlrDelayUnitInPro(1 downto 0);
llrduaCodeLlrDelayUnitOutPro(0) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
when "0100" =>
phaPhaseOffsetOutPro(nNumProcessors-1 downto 4) <= phaPhaseOffsetInIpIf(5 downto 0);
phaPhaseOffsetOutPro(4-1 downto 0)              <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 6);
llrduaInfoLlrDelayUnitOutPro(4) <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 5) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 4);
llrduaInfoLlrDelayUnitOutPro(3 downto 1) <= llrduaInfoLlrDelayUnitInPro(2 downto 0);
llrduaInfoLlrDelayUnitOutPro(0) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(4) <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 5) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 4);
llrduaCodeLlrDelayUnitOutPro(3 downto 1) <= llrduaCodeLlrDelayUnitInPro(2 downto 0);
llrduaCodeLlrDelayUnitOutPro(0) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
when "0101" =>
phaPhaseOffsetOutPro(nNumProcessors-1 downto 5) <= phaPhaseOffsetInIpIf(4 downto 0);
phaPhaseOffsetOutPro(5-1 downto 0)              <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 5);
llrduaInfoLlrDelayUnitOutPro(5) <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 6) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 5);
llrduaInfoLlrDelayUnitOutPro(4 downto 1) <= llrduaInfoLlrDelayUnitInPro(3 downto 0);
llrduaInfoLlrDelayUnitOutPro(0) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(5) <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 6) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 5);
llrduaCodeLlrDelayUnitOutPro(4 downto 1) <= llrduaCodeLlrDelayUnitInPro(3 downto 0);
llrduaCodeLlrDelayUnitOutPro(0) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
when "0110" =>
phaPhaseOffsetOutPro(nNumProcessors-1 downto 6) <= phaPhaseOffsetInIpIf(3 downto 0);
phaPhaseOffsetOutPro(6-1 downto 0)              <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 4);
llrduaInfoLlrDelayUnitOutPro(6) <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 7) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 6);
llrduaInfoLlrDelayUnitOutPro(5 downto 1) <= llrduaInfoLlrDelayUnitInPro(4 downto 0);
llrduaInfoLlrDelayUnitOutPro(0) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(6) <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 7) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 6);
llrduaCodeLlrDelayUnitOutPro(5 downto 1) <= llrduaCodeLlrDelayUnitInPro(4 downto 0);
llrduaCodeLlrDelayUnitOutPro(0) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
when "0111" =>
phaPhaseOffsetOutPro(nNumProcessors-1 downto 7) <= phaPhaseOffsetInIpIf(2 downto 0);
phaPhaseOffsetOutPro(7-1 downto 0)              <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 3);
llrduaInfoLlrDelayUnitOutPro(7) <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 8) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 7);
llrduaInfoLlrDelayUnitOutPro(6 downto 1) <= llrduaInfoLlrDelayUnitInPro(5 downto 0);
llrduaInfoLlrDelayUnitOutPro(0) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(7) <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 8) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 7);
llrduaCodeLlrDelayUnitOutPro(6 downto 1) <= llrduaCodeLlrDelayUnitInPro(5 downto 0);
llrduaCodeLlrDelayUnitOutPro(0) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
when "1000" =>
phaPhaseOffsetOutPro(nNumProcessors-1 downto 8) <= phaPhaseOffsetInIpIf(1 downto 0);
phaPhaseOffsetOutPro(8-1 downto 0)              <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 2);
llrduaInfoLlrDelayUnitOutPro(8) <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1 downto 9) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 8);
llrduaInfoLlrDelayUnitOutPro(7 downto 1) <= llrduaInfoLlrDelayUnitInPro(6 downto 0);
llrduaInfoLlrDelayUnitOutPro(0) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(8) <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1 downto 9) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-2 →
    ↪ downto 8);
llrduaCodeLlrDelayUnitOutPro(7 downto 1) <= llrduaCodeLlrDelayUnitInPro(6 downto 0);
llrduaCodeLlrDelayUnitOutPro(0) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
```

```
when "1001" =>
phaPhaseOffsetOutPro(nNumProcessors-1)                    <= phaPhaseOffsetInIpIf(0);
phaPhaseOffsetOutPro(nNumProcessors-2 downto 0)           <= phaPhaseOffsetInIpIf(nNumProcessors-1 downto 1);
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-1)            <= llrduInfoLlrDelayUnitInIpIf;
llrduaInfoLlrDelayUnitOutPro(nNumProcessors-2 downto 1) <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-3 →
    ↪ downto 0);
llrduaInfoLlrDelayUnitOutPro(0)                           <= llrduaInfoLlrDelayUnitInPro(nNumProcessors-1);
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-1)            <= llrduCodeLlrDelayUnitInIpIf;
llrduaCodeLlrDelayUnitOutPro(nNumProcessors-2 downto 1) <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-3 →
    ↪ downto 0);
llrduaCodeLlrDelayUnitOutPro(0)                           <= llrduaCodeLlrDelayUnitInPro(nNumProcessors-1);
when others =>
phaPhaseOffsetOutPro          <= PHASE_ARRAY_RESET;
llrduaInfoLlrDelayUnitOutPro <= LOG_LIKE_RATIO_DELAY_UNIT_ARRAY_RESET;
llrduaCodeLlrDelayUnitOutPro <= LOG_LIKE_RATIO_DELAY_UNIT_ARRAY_RESET;
end case;

  end process PROCESSOR_INPUTS;

-------------------------------------------------------------------------------
-- PROCESSOR_OUTPUTS
-------------------------------------------------------------------------------

  PROCESSOR_OUTPUTS : process ( slvControlFp, llrduaInfoLlrDelayUnitInPro,
                                llrduaCodeLlrDelayUnitInPro                ) is
  begin


case slvControlFp( 13 downto 8 ) is
when "000000" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(0)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(0)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(0)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(0)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(0)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(0)(0).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(0);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(0);
when "010000" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(0)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(0)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(0)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(0)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(0)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(0)(1).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(0);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(0);
when "100000" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(0)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(0)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(0)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(0)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(0)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(0)(2).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(0);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(0);
when "110000" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(0)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(0)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(0)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(0)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(0)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(0)(3).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(0);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(0);
when "000001" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(1)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(1)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(1)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(1)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(1)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(1)(0).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(1);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(1);
when "010001" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(1)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(1)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(1)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(1)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(1)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(1)(1).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(1);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(1);
```

```
when "100001" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(1)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(1)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(1)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(1)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(1)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(1)(2).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(1);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(1);
when "110001" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(1)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(1)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(1)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(1)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(1)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(1)(3).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(1);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(1);
when "000010" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(2)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(2)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(2)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(2)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(2)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(2)(0).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(2);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(2);
when "010010" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(2)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(2)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(2)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(2)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(2)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(2)(1).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(2);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(2);
when "100010" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(2)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(2)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(2)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(2)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(2)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(2)(2).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(2);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(2);
when "110010" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(2)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(2)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(2)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(2)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(2)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(2)(3).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(2);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(2);
when "000011" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(3)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(3)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(3)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(3)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(3)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(3)(0).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(3);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(3);
when "010011" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(3)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(3)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(3)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(3)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(3)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(3)(1).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(3);
llrduCodeLleDelayUnitOutSignEx    <= llrduaCodeLlrDelayUnitInPro(3);
when "100011" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )      <= llrduaInfoLlrDelayUnitInPro(3)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)         <= llrduaInfoLlrDelayUnitInPro(3)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                          <= llrduaInfoLlrDelayUnitInPro(3)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                     <= llrduaCodeLlrDelayUnitInPro(3)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                       <= llrduaCodeLlrDelayUnitInPro(3)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)            <= llrduaCodeLlrDelayUnitInPro(3)(2).Mag;
llrduInfoLleDelayUnitOutSignEx    <= llrduaInfoLlrDelayUnitInPro(3);
```

```
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(3);
when "110011" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(3)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(3)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(3)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(3)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(3)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(3)(3).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(3);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(3);
when "000100" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(4)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(4)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(4)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(4)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(4)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(4)(0).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(4);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(4);
when "010100" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(4)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(4)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(4)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(4)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(4)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(4)(1).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(4);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(4);
when "100100" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(4)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(4)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(4)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(4)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(4)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(4)(2).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(4);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(4);
when "110100" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(4)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(4)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(4)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(4)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(4)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(4)(3).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(4);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(4);
when "000101" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(5)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(5)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(5)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(5)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(5)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(5)(0).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(5);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(5);
when "010101" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(5)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(5)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(5)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(5)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(5)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(5)(1).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(5);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(5);
when "100101" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(5)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(5)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(5)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(5)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(5)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(5)(2).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(5);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(5);
when "110101" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(5)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(5)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(5)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(5)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(5)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(5)(3).Mag;
```

```
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(5);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(5);
when "000110" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(6)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(6)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(6)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(6)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(6)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(6)(0).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(6);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(6);
when "010110" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(6)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(6)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(6)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(6)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(6)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(6)(1).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(6);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(6);
when "100110" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(6)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(6)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(6)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(6)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(6)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(6)(2).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(6);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(6);
when "110110" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(6)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(6)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(6)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(6)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(6)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(6)(3).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(6);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(6);
when "000111" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(7)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(7)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(7)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(7)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(7)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(7)(0).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(7);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(7);
when "010111" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(7)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(7)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(7)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(7)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(7)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(7)(1).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(7);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(7);
when "100111" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(7)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(7)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(7)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(7)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(7)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(7)(2).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(7);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(7);
when "110111" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(7)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(7)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(7)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(7)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(7)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)             <= llrduaCodeLlrDelayUnitInPro(7)(3).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(7);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(7);
when "001000" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )       <= llrduaInfoLlrDelayUnitInPro(8)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)          <= llrduaInfoLlrDelayUnitInPro(8)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                           <= llrduaInfoLlrDelayUnitInPro(8)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                      <= llrduaCodeLlrDelayUnitInPro(8)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                        <= llrduaCodeLlrDelayUnitInPro(8)(0).Sgn;
```

```
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)              <= llrduaCodeLlrDelayUnitInPro(8)(0).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(8);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(8);
when "011000" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )        <= llrduaInfoLlrDelayUnitInPro(8)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)           <= llrduaInfoLlrDelayUnitInPro(8)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                            <= llrduaInfoLlrDelayUnitInPro(8)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                       <= llrduaCodeLlrDelayUnitInPro(8)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                         <= llrduaCodeLlrDelayUnitInPro(8)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)              <= llrduaCodeLlrDelayUnitInPro(8)(1).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(8);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(8);
when "101000" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )        <= llrduaInfoLlrDelayUnitInPro(8)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)           <= llrduaInfoLlrDelayUnitInPro(8)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                            <= llrduaInfoLlrDelayUnitInPro(8)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                       <= llrduaCodeLlrDelayUnitInPro(8)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                         <= llrduaCodeLlrDelayUnitInPro(8)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)              <= llrduaCodeLlrDelayUnitInPro(8)(2).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(8);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(8);
when "111000" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )        <= llrduaInfoLlrDelayUnitInPro(8)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)           <= llrduaInfoLlrDelayUnitInPro(8)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                            <= llrduaInfoLlrDelayUnitInPro(8)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                       <= llrduaCodeLlrDelayUnitInPro(8)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                         <= llrduaCodeLlrDelayUnitInPro(8)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)              <= llrduaCodeLlrDelayUnitInPro(8)(3).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(8);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(8);
when "001001" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )        <= llrduaInfoLlrDelayUnitInPro(9)(0).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)           <= llrduaInfoLlrDelayUnitInPro(9)(0).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                            <= llrduaInfoLlrDelayUnitInPro(9)(0).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                       <= llrduaCodeLlrDelayUnitInPro(9)(0).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                         <= llrduaCodeLlrDelayUnitInPro(9)(0).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)              <= llrduaCodeLlrDelayUnitInPro(9)(0).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(9);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(9);
when "011001" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )        <= llrduaInfoLlrDelayUnitInPro(9)(1).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)           <= llrduaInfoLlrDelayUnitInPro(9)(1).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                            <= llrduaInfoLlrDelayUnitInPro(9)(1).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                       <= llrduaCodeLlrDelayUnitInPro(9)(1).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                         <= llrduaCodeLlrDelayUnitInPro(9)(1).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)              <= llrduaCodeLlrDelayUnitInPro(9)(1).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(9);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(9);
when "101001" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )        <= llrduaInfoLlrDelayUnitInPro(9)(2).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)           <= llrduaInfoLlrDelayUnitInPro(9)(2).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                            <= llrduaInfoLlrDelayUnitInPro(9)(2).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                       <= llrduaCodeLlrDelayUnitInPro(9)(2).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                         <= llrduaCodeLlrDelayUnitInPro(9)(2).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)              <= llrduaCodeLlrDelayUnitInPro(9)(2).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(9);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(9);
when "111001" =>
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth+1 )        <= llrduaInfoLlrDelayUnitInPro(9)(3).Sat;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth)           <= llrduaInfoLlrDelayUnitInPro(9)(3).Sgn;
slvHighSpeedOutputsTpInt((nLlrWidth+2)+nLlrWidth-1 downto nLlrWidth+2)
                                                            <= llrduaInfoLlrDelayUnitInPro(9)(3).Mag;
slvHighSpeedOutputsTpInt(nLlrWidth+1)                       <= llrduaCodeLlrDelayUnitInPro(9)(3).Sat;
slvHighSpeedOutputsTpInt(nLlrWidth)                         <= llrduaCodeLlrDelayUnitInPro(9)(3).Sgn;
slvHighSpeedOutputsTpInt(nLlrWidth-1 downto 0)              <= llrduaCodeLlrDelayUnitInPro(9)(3).Mag;
llrduInfoLleDelayUnitOutSignEx     <= llrduaInfoLlrDelayUnitInPro(9);
llrduCodeLleDelayUnitOutSignEx     <= llrduaCodeLlrDelayUnitInPro(9);
when others =>
slvHighSpeedOutputsTpInt(11 downto 3) <= (others=>'0');
llrduInfoLleDelayUnitOutSignEx     <= LOG_LIKE_RATIO_DELAY_UNIT_RESET;
llrduCodeLleDelayUnitOutSignEx     <= LOG_LIKE_RATIO_DELAY_UNIT_RESET;
end case;


   end process PROCESSOR_OUTPUTS;

end Behavioral;
```

Listing B.9: LdpccProcessor.vhd

```
--    Filename: LdpccProcessor.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------
--
--  Description
--  -----------
--
-- This is the top-level of a processor unit for the LDPC-CC decoder.
--
-- ------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity LdpccProcessor is
    Port ( clkClock                 : in LogicBit;
           rstReset                 : in LogicBit;
           llrduInfoLlrDelayUnitIn  : in LogLikeRatioDelayUnit;
           llrduCodeLlrDelayUnitIn  : in LogLikeRatioDelayUnit;
           phPhaseIn                : in Phase;
           phPhaseOffset            : in Phase;
           --Outputs
           llrduInfoLlrDelayUnitOut : out LogLikeRatioDelayUnit;
           llrduCodeLlrDelayUnitOut : out LogLikeRatioDelayUnit
         );
end LdpccProcessor;

architecture Behavioral of LdpccProcessor is

-- ------------------------------------------------------------------
-- COMPONENTS
-- ------------------------------------------------------------------

    -- Components which we instantiate within the main
    -- block.

    component LdpccParityCheckNodeWithInf
    Port ( clkClock        : in LogicBit;
           rstReset        : in LogicBit;
           llrLlrTermIn1   : in LogLikeRatio;
           llrLlrTermIn2   : in LogLikeRatio;
           llrLlrTermIn3   : in LogLikeRatio;
           llrLlrTermIn4   : in LogLikeRatio;
           llrLlrTermIn5   : in LogLikeRatio;
           llrLlrTermIn6   : in LogLikeRatio;
           --Outputs
           llrLlrTermOut1  : out LogLikeRatio;
           llrLlrTermOut2  : out LogLikeRatio;
           llrLlrTermOut3  : out LogLikeRatio;
           llrLlrTermOut4  : out LogLikeRatio;
           llrLlrTermOut5  : out LogLikeRatio;
           llrLlrTermOut6  : out LogLikeRatio
         );
    end component;

    component LdpccLogLikeAdd is
     Port ( clkClock                 : in LogicBit;
            rstReset                 : in LogicBit;
            llrduInfoLlrDelayUnitIn  : in LogLikeRatioDelayUnit;
            llrduInfoLlrDelayUnitOut : out LogLikeRatioDelayUnit
          );
    end component;

-- ------------------------------------------------------------------
-- REGISTERS
-- ------------------------------------------------------------------

    -- Registered values which must have an _Next value
    -- to work well. As per Massana design rules.

    signal llrduInfoLlrDelayLine_Next : LogLikeRatioDelayLine;
    signal llrduInfoLlrDelayLine      : LogLikeRatioDelayLine;

    signal llrduCodeLlrDelayLine_Next : LogLikeRatioDelayLine;
    signal llrduCodeLlrDelayLine      : LogLikeRatioDelayLine;
```

```
  signal llrParityTermIn1_Next : LogLikeRatio;
  signal llrParityTermIn1      : LogLikeRatio;
  signal llrParityTermIn2_Next : LogLikeRatio;
  signal llrParityTermIn2      : LogLikeRatio;
  signal llrParityTermIn3_Next : LogLikeRatio;
  signal llrParityTermIn3      : LogLikeRatio;
  signal llrParityTermIn4_Next : LogLikeRatio;
  signal llrParityTermIn4      : LogLikeRatio;
  signal llrParityTermIn5_Next : LogLikeRatio;
  signal llrParityTermIn5      : LogLikeRatio;
  signal llrParityTermIn6_Next : LogLikeRatio;
  signal llrParityTermIn6      : LogLikeRatio;

    -- Delay line for the phase input.

  signal phPhaseIn_Del1_Next   : Phase;
  signal phPhaseIn_Del1        : Phase;
  signal phPhaseIn_Del2_Next   : Phase;
  signal phPhaseIn_Del2        : Phase;
  signal phPhaseIn_Del3_Next   : Phase;
  signal phPhaseIn_Del3        : Phase;
  signal phPhaseIn_Del4_Next   : Phase;
  signal phPhaseIn_Del4        : Phase;

  signal ph4DeMux : std_logic_vector(255 downto 0);


-- COMBINATORIAL


    -- Combinatorial signals which are not assigned to registers.

  signal llrParityTermOut1 : LogLikeRatio;
  signal llrParityTermOut2 : LogLikeRatio;
  signal llrParityTermOut3 : LogLikeRatio;
  signal llrParityTermOut4 : LogLikeRatio;
  signal llrParityTermOut5 : LogLikeRatio;
  signal llrParityTermOut6 : LogLikeRatio;

begin

-- INSTANTIATIONS

  -- The parity check node which performs sign min and includes +inf
  -- capabilities. Note its outputs are registered but inputs are not.

ParityCheckNode : LdpccParityCheckNodeWithInf
    port map
        ( clkClock       => clkClock,
          rstReset       => rstReset,
          llrLlrTermIn1  => llrParityTermIn1,
          llrLlrTermIn2  => llrParityTermIn2,
          llrLlrTermIn3  => llrParityTermIn3,
          llrLlrTermIn4  => llrParityTermIn4,
          llrLlrTermIn5  => llrParityTermIn5,
          llrLlrTermIn6  => llrParityTermIn6,
          --Outputs
          llrLlrTermOut1  => llrParityTermOut1,
          llrLlrTermOut2  => llrParityTermOut2,
          llrLlrTermOut3  => llrParityTermOut3,
          llrLlrTermOut4  => llrParityTermOut4,
          llrLlrTermOut5  => llrParityTermOut5,
          llrLlrTermOut6  => llrParityTermOut6
        );

  -- The LLR addition block that performs the addition at the final stage of the
  -- adder.

LlrInfoAdder : LdpccLogLikeAdd
    Port map
        ( clkClock                 => clkClock,
          rstReset                 => rstReset,
          llrduInfoLlrDelayUnitIn  => llrduInfoLlrDelayLine(llrduInfoLlrDelayLine'left(1)),
          llrduInfoLlrDelayUnitOut => llrduInfoLlrDelayUnitOut
        );

LlrCodeAdder : LdpccLogLikeAdd
    Port map
        ( clkClock                 => clkClock,
          rstReset                 => rstReset,
          llrduInfoLlrDelayUnitIn  => llrduCodeLlrDelayLine(llrduCodeLlrDelayLine'left(1)),
          llrduInfoLlrDelayUnitOut => llrduCodeLlrDelayUnitOut
        );


-- PHASE_IN_DELAY_LINE
```

```
PHASE_IN_DELAY_LINE : process ( phPhaseIn , phPhaseIn_Del1 , phPhaseIn_Del2 ,
                                phPhaseIn_Del3 , phPhaseOffset ) is
begin

   -- We must add the offset to the input phase and then correct to
   -- ensure it does not sit outside the range of phases for this code.

  if ( phPhaseOffset > phPhaseIn ) then
    phPhaseIn_Del1_Next <= PHASE_MAXIMUM-phPhaseOffset+phPhaseIn+"00000001";
  else
    phPhaseIn_Del1_Next <= phPhaseIn - phPhaseOffset;
  end if;

  phPhaseIn_Del2_Next <= phPhaseIn_Del1;
  phPhaseIn_Del3_Next <= phPhaseIn_Del2;
  phPhaseIn_Del4_Next <= phPhaseIn_Del3;

end process PHASE_IN_DELAY_LINE;
```

```
-- CLOCK_UPDATE
```

```
   -- This is the process which updates all the registers
   -- with their new values.

  CLOCK_UPDATE : process ( clkClock ) is
  begin

    if ( clkClock'event and clkClock='1' ) then

      if ( rstReset = '1' ) then

        llrduInfoLlrDelayLine <= LOG_LIKE_RATIO_DELAY_LINE_RESET;
       llrduCodeLlrDelayLine <= LOG_LIKE_RATIO_DELAY_LINE_RESET;

        llrParityTermIn1 <= LOG_LIKE_RATIO_RESET;
        llrParityTermIn2 <= LOG_LIKE_RATIO_RESET;
        llrParityTermIn3 <= LOG_LIKE_RATIO_RESET;
        llrParityTermIn4 <= LOG_LIKE_RATIO_RESET;
        llrParityTermIn5 <= LOG_LIKE_RATIO_RESET;
        llrParityTermIn6 <= LOG_LIKE_RATIO_RESET;

        phPhaseIn_Del1   <= (others=>'0');
        phPhaseIn_Del2   <= (others=>'0');
        phPhaseIn_Del3   <= (others=>'0');
        phPhaseIn_Del4   <= (others=>'0');

      else

        llrduInfoLlrDelayLine <= llrduInfoLlrDelayLine_Next;
        llrduCodeLlrDelayLine <= llrduCodeLlrDelayLine_Next;

        llrParityTermIn1 <= llrParityTermIn1_Next;
        llrParityTermIn2 <= llrParityTermIn2_Next;
        llrParityTermIn3 <= llrParityTermIn3_Next;
        llrParityTermIn4 <= llrParityTermIn4_Next;
        llrParityTermIn5 <= llrParityTermIn5_Next;
        llrParityTermIn6 <= llrParityTermIn6_Next;

        phPhaseIn_Del1   <= phPhaseIn_Del1_Next;
        phPhaseIn_Del2   <= phPhaseIn_Del2_Next;
        phPhaseIn_Del3   <= phPhaseIn_Del3_Next;
        phPhaseIn_Del4   <= phPhaseIn_Del4_Next;


    end if;

     end if;

  end process CLOCK_UPDATE;

-- syn time for this process by itself: ~15min
process (phPhaseIn_Del4) is
    variable ph4DeMux0 : std_logic_vector(1 downto 0);
    variable ph4DeMux1 : std_logic_vector(3 downto 0);
    variable ph4DeMux2 : std_logic_vector(7 downto 0);
    variable ph4DeMux3 : std_logic_vector(15 downto 0);
    variable ph4DeMux4 : std_logic_vector(31 downto 0);
    variable ph4DeMux5 : std_logic_vector(63 downto 0);
    variable ph4DeMux6 : std_logic_vector(127 downto 0);
    variable ph4DeMux7 : std_logic_vector(255 downto 0);

begin

    if phPhaseIn_Del4(0) = '1' then
    ph4DeMux0(0) := '0';
```

```
    ph4DeMux0(1) := '1';
    else
    ph4DeMux0(0) := '1';
    ph4DeMux0(1) := '0';
    end if;

    if phPhaseIn_Del4(1) = '1' then
    ph4DeMux1(1 downto 0) := (others=>'0');
    ph4DeMux1(3 downto 2) := ph4DeMux0;
    else
    ph4DeMux1(1 downto 0) := ph4DeMux0;
    ph4DeMux1(3 downto 2) := (others=>'0');
    end if;

    if phPhaseIn_Del4(2) = '1' then
    ph4DeMux2(3 downto 0) := (others=>'0');
    ph4DeMux2(7 downto 4) := ph4DeMux1;
    else
    ph4DeMux2(3 downto 0) := ph4DeMux1;
    ph4DeMux2(7 downto 4) := (others=>'0');
    end if;

    if phPhaseIn_Del4(3) = '1' then
    ph4DeMux3(7 downto 0) := (others=>'0');
    ph4DeMux3(15 downto 8) := ph4DeMux2;
    else
    ph4DeMux3(7 downto 0) := ph4DeMux2;
    ph4DeMux3(15 downto 8) := (others=>'0');
    end if;

    if phPhaseIn_Del4(4) = '1' then
    ph4DeMux4(15 downto 0) := (others=>'0');
    ph4DeMux4(31 downto 16) := ph4DeMux3;
    else
    ph4DeMux4(15 downto 0) := ph4DeMux3;
    ph4DeMux4(31 downto 16) := (others=>'0');
    end if;

    if phPhaseIn_Del4(5) = '1' then
    ph4DeMux5(31 downto 0) := (others=>'0');
    ph4DeMux5(63 downto 32) := ph4DeMux4;
    else
    ph4DeMux5(31 downto 0) := ph4DeMux4;
    ph4DeMux5(63 downto 32) := (others=>'0');
    end if;

    if phPhaseIn_Del4(6) = '1' then
    ph4DeMux6(63 downto 0) := (others=>'0');
    ph4DeMux6(127 downto 64) := ph4DeMux5;
    else
    ph4DeMux6(63 downto 0) := ph4DeMux5;
    ph4DeMux6(127 downto 64) := (others=>'0');
    end if;

    if phPhaseIn_Del4(7) = '1' then
    ph4DeMux7(127 downto 0) := (others=>'0');
    ph4DeMux7(255 downto 128) := ph4DeMux6;
    else
    ph4DeMux7(127 downto 0) := ph4DeMux6;
    ph4DeMux7(255 downto 128) := (others=>'0');
    end if;

    ph4DeMux <= ph4DeMux7;

end process;


-- GET_WRITE_PARITY_CHECK_OUTPUTS_AND_DELAY_UPDATE

  -- elaborate time of this process by itself: ~15min
  GET_WRITE_PARITY_CHECK_OUTPUTS_AND_DELAY_UPDATE :
    process ( llrduInfoLlrDelayUnitIn , llrduCodeLlrDelayUnitIn ) is
    begin
          -- Place the most recent Information bit and code bit into
          -- first point of the delay line (entry 0)

       llrduInfoLlrDelayLine_Next(0) <= llrduInfoLlrDelayUnitIn;
       llrduCodeLlrDelayLine_Next(0) <= llrduCodeLlrDelayUnitIn;

    end process;

    process ( llrduCodeLlrDelayLine, llrduInfoLlrDelayLine, phPhaseIn_Del1 ) is
    begin

          -- Place the most recent Information bit and code bit into
          -- first point of the delay line (entry 0)

        -- Now comes the tricky part. We need to assign the parity check (PC)
```

```
-- inputs from the delay lines and this varies from one phase to
-- the next. We use an array of PC elements to make the code clean.
-- NB Since 2 of the code inputs are fixed the middle one always has weight
-- 2. This simplifies the code for the code delay line a lot. The info
-- addressing is more complex.

-- The last codebit and most recent codebit are always used so we
-- can hardcode these here. Note we always register the input first
-- to be nice for synthesis.

-- Autogenerated code for LDPCC code ldpcc_128_1_2_3_6.pcm.
llrParityTermIn1_Next <= llrduCodeLlrDelayLine(0)(1);
llrParityTermIn2_Next <= llrduCodeLlrDelayLine(nCodeMemory-1)(3);
case phPhaseIn_Del1 is
when "00000000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(11)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(8)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(65)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(105)(3);
when "00000001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(9)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(32)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(53)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(121)(3);
when "00000010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(29)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(40)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(62)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(108)(3);
when "00000011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(20)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(12)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(73)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(103)(3);
when "00000100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(38)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(25)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(57)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(113)(3);
when "00000101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(28)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(9)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(77)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(115)(3);
when "00000110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(22)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(38)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(60)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(125)(3);
when "00000111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(31)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(2)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(30)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(70)(2);
when "00001000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(34)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(22)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(79)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(95)(3);
when "00001001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(39)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(24)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(55)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(126)(3);
when "00001010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(42)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(52)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(74)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(125)(3);
when "00001011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(15)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(11)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(52)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(89)(2);
when "00001100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(25)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(19)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(68)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(116)(3);
when "00001101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(22)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(7)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(33)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(81)(2);
when "00001110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(15)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(41)(2);
```

```
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(88)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(111)(3);
when "00001111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(51)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(44)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(95)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(126)(3);
when "00010000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(31)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(35)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(99)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(119)(3);
when "00010001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(35)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(35)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(78)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(109)(3);
when "00010010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(21)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(43)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(97)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(125)(3);
when "00010011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(58)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(21)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(66)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(110)(3);
when "00010100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(39)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(7)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(25)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(78)(3);
when "00010101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(43)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(0)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(51)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(90)(3);
when "00010110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(64)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(28)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(77)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(120)(3);
when "00010111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(44)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(49)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(100)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(125)(3);
when "00011000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(36)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(40)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(106)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(120)(3);
when "00011001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(53)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(23)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(68)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(87)(3);
when "00011010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(63)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(50)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(93)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(125)(3);
when "00011011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(62)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(62)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(86)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(115)(3);
when "00011100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(68)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(12)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(39)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(94)(2);
when "00011101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(49)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(14)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(42)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(77)(2);
when "00011110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(63)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(33)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(105)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(123)(3);
when "00011111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(72)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(24)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(67)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(88)(2);
when "00100000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(61)(2);
```

```
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(29)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(69)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(117)(3);
when "00100001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(76)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(25)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(34)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(114)(3);
when "00100010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(44)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(51)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(120)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(123)(3);
when "00100011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(60)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(31)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(68)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(80)(2);
when "00100100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(43)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(8)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(76)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(109)(3);
when "00100101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(75)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(20)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(47)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(86)(2);
when "00100110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(44)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(3)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(77)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(88)(2);
when "00100111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(53)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(0)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(61)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(123)(3);
when "00101000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(71)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(6)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(74)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(91)(2);
when "00101001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(46)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(30)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(53)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(85)(2);
when "00101010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(44)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(41)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(70)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(118)(2);
when "00101011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(27)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(27)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(47)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(111)(3);
when "00101100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(37)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(2)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(45)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(122)(3);
when "00101101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(11)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(33)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(70)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(92)(3);
when "00101110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(11)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(2)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(46)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(80)(3);
when "00101111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(41)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(10)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(57)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(98)(3);
when "00110000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(15)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(40)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(64)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(98)(3);
when "00110001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(32)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(29)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(54)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(114)(3);
when "00110010" =>
```

```
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(14)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(24)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(58)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(126)(3);
when "00110011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(48)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(17)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(57)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(110)(3);
when "00110100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(47)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(30)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(84)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(100)(3);
when "00110101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(53)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(30)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(68)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(116)(3);
when "00110110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(33)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(19)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(81)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(121)(3);
when "00110111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(28)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(6)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(14)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(72)(2);
when "00111000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(34)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(10)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(35)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(82)(3);
when "00111001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(53)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(18)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(93)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(117)(3);
when "00111010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(33)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(30)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(97)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(114)(3);
when "00111011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(35)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(23)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(88)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(125)(3);
when "00111100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(51)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(1)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(42)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(103)(3);
when "00111101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(49)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(58)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(89)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(107)(3);
when "00111110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(20)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(31)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(75)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(123)(3);
when "00111111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(49)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(30)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(77)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(112)(3);
when "01000000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(44)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(7)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(59)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(99)(2);
when "01000001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(36)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(17)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(38)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(98)(2);
when "01000010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(58)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(60)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(106)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(118)(3);
when "01000011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(41)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(65)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(89)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(124)(3);
```

```
when "01000100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(40)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(18)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(59)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(109)(3);
when "01000101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(54)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(15)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(50)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(100)(3);
when "01000110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(30)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(30)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(72)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(115)(3);
when "01000111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(34)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(42)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(78)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(124)(3);
when "01001000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(71)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(42)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(109)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(116)(3);
when "01001001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(42)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(66)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(103)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(127)(3);
when "01001010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(42)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(16)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(61)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(86)(3);
when "01001011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(73)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(4)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(43)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(84)(2);
when "01001100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(35)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(24)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(51)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(99)(3);
when "01001101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(58)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(3)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(73)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(80)(2);
when "01001110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(48)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(22)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(54)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(116)(3);
when "01001111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(69)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(34)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(68)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(100)(3);
when "01010000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(62)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(37)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(65)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(98)(2);
when "01010001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(70)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(21)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(67)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(101)(3);
when "01010010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(44)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(27)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(65)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(106)(3);
when "01010011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(60)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(16)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(45)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(94)(2);
when "01010100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(45)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(19)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(74)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(103)(3);
when "01010101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(72)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(38)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(84)(2);
```

```
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(127)(3);
when "01010110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(18)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(41)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(63)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(119)(3);
when "01010111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(23)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(40)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(66)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(97)(3);
when "01011000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(33)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(8)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(70)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(113)(3);
when "01011001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(9)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(21)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(78)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(124)(3);
when "01011010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(43)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(18)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(62)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(107)(3);
when "01011011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(40)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(26)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(83)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(96)(3);
when "01011100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(13)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(8)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(61)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(93)(3);
when "01011101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(43)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(26)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(87)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(109)(3);
when "01011110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(31)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(5)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(31)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(80)(2);
when "01011111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(13)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(10)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(88)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(103)(3);
when "01100000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(11)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(20)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(91)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(110)(3);
when "01100001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(44)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(43)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(85)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(103)(3);
when "01100010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(46)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(25)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(68)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(120)(3);
when "01100011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(41)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(35)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(65)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(106)(3);
when "01100100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(29)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(1)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(49)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(71)(2);
when "01100101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(31)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(27)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(81)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(119)(3);
when "01100110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(56)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(16)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(31)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(93)(2);
when "01100111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(25)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(44)(2);
```

```
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(63)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(105)(3);
when "01101000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(61)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(56)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(71)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(119)(3);
when "01101001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(30)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(52)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(92)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(116)(3);
when "01101010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(49)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(13)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(24)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(82)(2);
when "01101011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(33)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(10)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(50)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(104)(3);
when "01101100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(35)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(5)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(58)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(93)(3);
when "01101101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(50)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(8)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(63)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(90)(2);
when "01101110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(43)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(67)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(106)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(123)(3);
when "01101111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(35)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(11)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(34)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(89)(2);
when "01110000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(51)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(3)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(51)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(112)(3);
when "01110001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(44)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(21)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(30)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(103)(2);
when "01110010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(52)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(65)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(79)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(117)(3);
when "01110011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(55)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(5)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(45)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(77)(2);
when "01110100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(60)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(60)(2);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(89)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(125)(3);
when "01110101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(52)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(9)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(62)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(101)(3);
when "01110110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(46)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(12)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(56)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(79)(3);
when "01110111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(74)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(17)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(75)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(83)(2);
when "01111000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(54)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(22)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(60)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(83)(2);
when "01111001" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(37)(2);
```

```
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(55)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(120)(3);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(125)(3);
when "01111010" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(78)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(31)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(41)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(81)(2);
when "01111011" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(42)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(6)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(54)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(121)(3);
when "01111100" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(47)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(36)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(66)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(107)(3);
when "01111101" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(71)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(38)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(47)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(99)(2);
when "01111110" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(43)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(31)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(47)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(94)(2);
when "01111111" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(78)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(22)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(52)(1);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(85)(2);
when "10000000" =>
llrParityTermIn3_Next <= llrduCodeLlrDelayLine(80)(2);
llrParityTermIn4_Next <= llrduInfoLlrDelayLine(38)(1);
llrParityTermIn5_Next <= llrduInfoLlrDelayLine(76)(2);
llrParityTermIn6_Next <= llrduInfoLlrDelayLine(103)(2);
when others =>
llrParityTermIn3_Next <= LOG_LIKE_RATIO_RESET;
llrParityTermIn4_Next <= LOG_LIKE_RATIO_RESET;
llrParityTermIn5_Next <= LOG_LIKE_RATIO_RESET;
llrParityTermIn6_Next <= LOG_LIKE_RATIO_RESET;
end case;
end process;


-- elaborate time by itself: 8 min
    process ( llrduInfoLlrDelayLine, ph4DeMux,
        llrParityTermOut4 , llrParityTermOut5 , llrParityTermOut6  ) is

   begin
-- TB
llrduInfoLlrDelayLine_Next(139 downto 1) <= llrduInfoLlrDelayLine(138 downto 0);

  -- Code for Info Registers 4
if ( ph4DeMux(21) = '1' ) then
llrduInfoLlrDelayLine_Next(4)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(39) = '1' ) then
llrduInfoLlrDelayLine_Next(4)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 5
if ( ph4DeMux(60) = '1' ) then
llrduInfoLlrDelayLine_Next(5)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(100) = '1' ) then
llrduInfoLlrDelayLine_Next(5)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 6
if ( ph4DeMux(7) = '1' ) then
llrduInfoLlrDelayLine_Next(6)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(44) = '1' ) then
llrduInfoLlrDelayLine_Next(6)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(46) = '1' ) then
llrduInfoLlrDelayLine_Next(6)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 7
if ( ph4DeMux(38) = '1' ) then
llrduInfoLlrDelayLine_Next(7)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(77) = '1' ) then
llrduInfoLlrDelayLine_Next(7)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(112) = '1' ) then
llrduInfoLlrDelayLine_Next(7)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 8
if ( ph4DeMux(75) = '1' ) then
```

```
llrduInfoLlrDelayLine_Next(8)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 9
if ( ph4DeMux(94) = '1' ) then
llrduInfoLlrDelayLine_Next(9)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(108) = '1' ) then
llrduInfoLlrDelayLine_Next(9)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(115) = '1' ) then
llrduInfoLlrDelayLine_Next(9)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 10
if ( ph4DeMux(40) = '1' ) then
llrduInfoLlrDelayLine_Next(10)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(55) = '1' ) then
llrduInfoLlrDelayLine_Next(10)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(123) = '1' ) then
llrduInfoLlrDelayLine_Next(10)(1) <= llrParityTermOut4;
end if;


  -- Code for Info Registers 11
if ( ph4DeMux(13) = '1' ) then
llrduInfoLlrDelayLine_Next(11)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(20) = '1' ) then
llrduInfoLlrDelayLine_Next(11)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(64) = '1' ) then
llrduInfoLlrDelayLine_Next(11)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 12

if ( ph4DeMux(0) = '1' ) then
llrduInfoLlrDelayLine_Next(12)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(36) = '1' ) then
llrduInfoLlrDelayLine_Next(12)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(88) = '1' ) then
llrduInfoLlrDelayLine_Next(12)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(92) = '1' ) then
llrduInfoLlrDelayLine_Next(12)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(109) = '1' ) then
llrduInfoLlrDelayLine_Next(12)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 13
if ( ph4DeMux(5) = '1' ) then
llrduInfoLlrDelayLine_Next(13)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(117) = '1' ) then
llrduInfoLlrDelayLine_Next(13)(1) <= llrParityTermOut4;
end if;

 -- Code for Info Registers 14
if ( ph4DeMux(47) = '1' ) then
llrduInfoLlrDelayLine_Next(14)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(56) = '1' ) then
llrduInfoLlrDelayLine_Next(14)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(95) = '1' ) then
llrduInfoLlrDelayLine_Next(14)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(107) = '1' ) then
llrduInfoLlrDelayLine_Next(14)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 15

if ( ph4DeMux(11) = '1' ) then
llrduInfoLlrDelayLine_Next(15)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(111) = '1' ) then
llrduInfoLlrDelayLine_Next(15)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 16
if ( ph4DeMux(3) = '1' ) then
llrduInfoLlrDelayLine_Next(16)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(28) = '1' ) then
llrduInfoLlrDelayLine_Next(16)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(118) = '1' ) then
llrduInfoLlrDelayLine_Next(16)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 17
if ( ph4DeMux(106) = '1' ) then
llrduInfoLlrDelayLine_Next(17)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 18
if ( ph4DeMux(29) = '1' ) then
llrduInfoLlrDelayLine_Next(18)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(55) = '1' ) then
```

```
llrduInfoLlrDelayLine_Next(18)(1) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 19
if ( ph4DeMux(69) = '1' ) then
llrduInfoLlrDelayLine_Next(19)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 20
if ( ph4DeMux(74) = '1' ) then
llrduInfoLlrDelayLine_Next(20)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(83) = '1' ) then
llrduInfoLlrDelayLine_Next(20)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(102) = '1' ) then
llrduInfoLlrDelayLine_Next(20)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 21
if ( ph4DeMux(65) = '1' ) then
llrduInfoLlrDelayLine_Next(21)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(119) = '1' ) then
llrduInfoLlrDelayLine_Next(21)(1) <= llrParityTermOut4;
end if;
if ( ph4DeMux(51) = '1' ) then
llrduInfoLlrDelayLine_Next(21)(2) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 22
if ( ph4DeMux(68) = '1' ) then
llrduInfoLlrDelayLine_Next(22)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(90) = '1' ) then
llrduInfoLlrDelayLine_Next(22)(1) <= llrParityTermOut4;
end if;
if ( ph4DeMux(57) = '1' ) then
llrduInfoLlrDelayLine_Next(22)(2) <= llrParityTermOut4;
end if;

 -- Code for Info Registers 23

if ( ph4DeMux(12) = '1' ) then
llrduInfoLlrDelayLine_Next(23)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(84) = '1' ) then
llrduInfoLlrDelayLine_Next(23)(1) <= llrParityTermOut4;
end if;
if ( ph4DeMux(54) = '1' ) then
llrduInfoLlrDelayLine_Next(23)(2) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 24
if ( ph4DeMux(37) = '1' ) then
llrduInfoLlrDelayLine_Next(24)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(96) = '1' ) then
llrduInfoLlrDelayLine_Next(24)(1) <= llrParityTermOut4;
end if;

  -- Code for Info Registers 25

if ( ph4DeMux(19) = '1' ) then
llrduInfoLlrDelayLine_Next(25)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(81) = '1' ) then
llrduInfoLlrDelayLine_Next(25)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(89) = '1' ) then
llrduInfoLlrDelayLine_Next(25)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(113) = '1' ) then
llrduInfoLlrDelayLine_Next(25)(1) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 26

if ( ph4DeMux(8) = '1' ) then
llrduInfoLlrDelayLine_Next(26)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(78) = '1' ) then
llrduInfoLlrDelayLine_Next(26)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(120) = '1' ) then
llrduInfoLlrDelayLine_Next(26)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(127) = '1' ) then
llrduInfoLlrDelayLine_Next(26)(1) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 27

if ( ph4DeMux(25) = '1' ) then
llrduInfoLlrDelayLine_Next(27)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(59) = '1' ) then
llrduInfoLlrDelayLine_Next(27)(1) <= llrParityTermOut4;

end if;
```

```
  -- Code for Info Registers 28

if ( ph4DeMux(9) = '1' ) then
llrduInfoLlrDelayLine_Next(28)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(31) = '1' ) then
llrduInfoLlrDelayLine_Next(28)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(50) = '1' ) then
llrduInfoLlrDelayLine_Next(28)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(76) = '1' ) then
llrduInfoLlrDelayLine_Next(28)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(106) = '1' ) then
llrduInfoLlrDelayLine_Next(28)(1) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 29

if ( ph4DeMux(20) = '1' ) then
llrduInfoLlrDelayLine_Next(29)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(33) = '1' ) then
llrduInfoLlrDelayLine_Next(29)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(98) = '1' ) then
llrduInfoLlrDelayLine_Next(29)(1) <= llrParityTermOut4;

end if;
if ( ph4DeMux(4) = '1' ) then
llrduInfoLlrDelayLine_Next(29)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 30

if ( ph4DeMux(91) = '1' ) then
llrduInfoLlrDelayLine_Next(30)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(93) = '1' ) then
llrduInfoLlrDelayLine_Next(30)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 31

if ( ph4DeMux(82) = '1' ) then
llrduInfoLlrDelayLine_Next(31)(1) <= llrParityTermOut4;


end if;
if ( ph4DeMux(43) = '1' ) then
llrduInfoLlrDelayLine_Next(31)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(101) = '1' ) then
llrduInfoLlrDelayLine_Next(31)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 32

if ( ph4DeMux(22) = '1' ) then
llrduInfoLlrDelayLine_Next(32)(1) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 33

if ( ph4DeMux(32) = '1' ) then
llrduInfoLlrDelayLine_Next(33)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(49) = '1' ) then
llrduInfoLlrDelayLine_Next(33)(1) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 34

if ( ph4DeMux(41) = '1' ) then
llrduInfoLlrDelayLine_Next(34)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(52) = '1' ) then
llrduInfoLlrDelayLine_Next(34)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(53) = '1' ) then
llrduInfoLlrDelayLine_Next(34)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(63) = '1' ) then
llrduInfoLlrDelayLine_Next(34)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(70) = '1' ) then
llrduInfoLlrDelayLine_Next(34)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(113) = '1' ) then
llrduInfoLlrDelayLine_Next(34)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(7) = '1' ) then
llrduInfoLlrDelayLine_Next(34)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(58) = '1' ) then
```

```
llrduInfoLlrDelayLine_Next(34)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 35

if ( ph4DeMux(35) = '1' ) then
llrduInfoLlrDelayLine_Next(35)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(62) = '1' ) then
llrduInfoLlrDelayLine_Next(35)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(94) = '1' ) then
llrduInfoLlrDelayLine_Next(35)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(122) = '1' ) then
llrduInfoLlrDelayLine_Next(35)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(126) = '1' ) then
llrduInfoLlrDelayLine_Next(35)(1) <= llrParityTermOut4;

end if;
if ( ph4DeMux(102) = '1' ) then
llrduInfoLlrDelayLine_Next(35)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 36

if ( ph4DeMux(1) = '1' ) then
llrduInfoLlrDelayLine_Next(36)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 37

if ( ph4DeMux(30) = '1' ) then
llrduInfoLlrDelayLine_Next(37)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(45) = '1' ) then
llrduInfoLlrDelayLine_Next(37)(1) <= llrParityTermOut4;

end if;
if ( ph4DeMux(13) = '1' ) then
llrduInfoLlrDelayLine_Next(37)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 38

if ( ph4DeMux(33) = '1' ) then
llrduInfoLlrDelayLine_Next(38)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(79) = '1' ) then
llrduInfoLlrDelayLine_Next(38)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(111) = '1' ) then
llrduInfoLlrDelayLine_Next(38)(1) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 39

if ( ph4DeMux(17) = '1' ) then
llrduInfoLlrDelayLine_Next(39)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(99) = '1' ) then
llrduInfoLlrDelayLine_Next(39)(1) <= llrParityTermOut4;

end if;
if ( ph4DeMux(16) = '1' ) then
llrduInfoLlrDelayLine_Next(39)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(56) = '1' ) then
llrduInfoLlrDelayLine_Next(39)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 40

if ( ph4DeMux(124) = '1' ) then
llrduInfoLlrDelayLine_Next(40)(1) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 41

if ( ph4DeMux(80) = '1' ) then
llrduInfoLlrDelayLine_Next(41)(1) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 42

if ( ph4DeMux(65) = '1' ) then
llrduInfoLlrDelayLine_Next(42)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(85) = '1' ) then
llrduInfoLlrDelayLine_Next(42)(1) <= llrParityTermOut4;
```

```
elsif ( ph4DeMux(125) = '1' ) then
llrduInfoLlrDelayLine_Next(42)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(128) = '1' ) then
llrduInfoLlrDelayLine_Next(42)(1) <= llrParityTermOut4;

end if;
if ( ph4DeMux(6) = '1' ) then
llrduInfoLlrDelayLine_Next(42)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 43

if ( ph4DeMux(28) = '1' ) then
llrduInfoLlrDelayLine_Next(43)(1) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 44

if ( ph4DeMux(24) = '1' ) then
llrduInfoLlrDelayLine_Next(44)(1) <= llrParityTermOut4;

end if;
if ( ph4DeMux(2) = '1' ) then
llrduInfoLlrDelayLine_Next(44)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(48) = '1' ) then
llrduInfoLlrDelayLine_Next(44)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(87) = '1' ) then
llrduInfoLlrDelayLine_Next(44)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 45

if ( ph4DeMux(42) = '1' ) then
llrduInfoLlrDelayLine_Next(45)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(122) = '1' ) then
llrduInfoLlrDelayLine_Next(45)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(14) = '1' ) then
llrduInfoLlrDelayLine_Next(45)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(86) = '1' ) then
llrduInfoLlrDelayLine_Next(45)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 46

if ( ph4DeMux(29) = '1' ) then
llrduInfoLlrDelayLine_Next(46)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(60) = '1' ) then
llrduInfoLlrDelayLine_Next(46)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(71) = '1' ) then
llrduInfoLlrDelayLine_Next(46)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(72) = '1' ) then
llrduInfoLlrDelayLine_Next(46)(1) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 47

if ( ph4DeMux(18) = '1' ) then
llrduInfoLlrDelayLine_Next(47)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(75) = '1' ) then
llrduInfoLlrDelayLine_Next(47)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(97) = '1' ) then
llrduInfoLlrDelayLine_Next(47)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 48

if ( ph4DeMux(15) = '1' ) then
llrduInfoLlrDelayLine_Next(48)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(103) = '1' ) then
llrduInfoLlrDelayLine_Next(48)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 49

if ( ph4DeMux(83) = '1' ) then
llrduInfoLlrDelayLine_Next(49)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(115) = '1' ) then
llrduInfoLlrDelayLine_Next(49)(1) <= llrParityTermOut5;
```

```
end if;
if ( ph4DeMux(44) = '1' ) then
llrduInfoLlrDelayLine_Next(49)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 50

if ( ph4DeMux(46) = '1' ) then
llrduInfoLlrDelayLine_Next(50)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 51

if ( ph4DeMux(37) = '1' ) then
llrduInfoLlrDelayLine_Next(51)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(125) = '1' ) then
llrduInfoLlrDelayLine_Next(51)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(126) = '1' ) then
llrduInfoLlrDelayLine_Next(51)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(43) = '1' ) then
llrduInfoLlrDelayLine_Next(51)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 52


  -- Code for Info Registers 53

if ( ph4DeMux(100) = '1' ) then
llrduInfoLlrDelayLine_Next(53)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(23) = '1' ) then
llrduInfoLlrDelayLine_Next(53)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 54

if ( ph4DeMux(69) = '1' ) then
llrduInfoLlrDelayLine_Next(54)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(26) = '1' ) then
llrduInfoLlrDelayLine_Next(54)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(107) = '1' ) then
llrduInfoLlrDelayLine_Next(54)(2) <= llrParityTermOut5;

end if;


  -- Code for Info Registers 55

if ( ph4DeMux(34) = '1' ) then
llrduInfoLlrDelayLine_Next(55)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(76) = '1' ) then
llrduInfoLlrDelayLine_Next(55)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(112) = '1' ) then
llrduInfoLlrDelayLine_Next(55)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(21) = '1' ) then
llrduInfoLlrDelayLine_Next(55)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 56

if ( ph4DeMux(105) = '1' ) then
llrduInfoLlrDelayLine_Next(56)(1) <= llrParityTermOut4;
elsif ( ph4DeMux(127) = '1' ) then
llrduInfoLlrDelayLine_Next(56)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(10) = '1' ) then
llrduInfoLlrDelayLine_Next(56)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(11) = '1' ) then
llrduInfoLlrDelayLine_Next(56)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 57
```

```
if ( ph4DeMux(1) = '1' ) then
llrduInfoLlrDelayLine_Next(57)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(41) = '1' ) then
llrduInfoLlrDelayLine_Next(57)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 58

if ( ph4DeMux(78) = '1' ) then
llrduInfoLlrDelayLine_Next(58)(1) <= llrParityTermOut5;
elsif ( ph4DeMux(123) = '1' ) then
llrduInfoLlrDelayLine_Next(58)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(49) = '1' ) then
llrduInfoLlrDelayLine_Next(58)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 59

if ( ph4DeMux(121) = '1' ) then
llrduInfoLlrDelayLine_Next(59)(1) <= llrParityTermOut4;

end if;
if ( ph4DeMux(9) = '1' ) then
llrduInfoLlrDelayLine_Next(59)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 60

if ( ph4DeMux(118) = '1' ) then
llrduInfoLlrDelayLine_Next(60)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(104) = '1' ) then
llrduInfoLlrDelayLine_Next(60)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 61

if ( ph4DeMux(4) = '1' ) then
llrduInfoLlrDelayLine_Next(61)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(47) = '1' ) then
llrduInfoLlrDelayLine_Next(61)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(51) = '1' ) then
llrduInfoLlrDelayLine_Next(61)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 62

if ( ph4DeMux(50) = '1' ) then
llrduInfoLlrDelayLine_Next(62)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(61) = '1' ) then
llrduInfoLlrDelayLine_Next(62)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(108) = '1' ) then
llrduInfoLlrDelayLine_Next(62)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 63

if ( ph4DeMux(68) = '1' ) then
llrduInfoLlrDelayLine_Next(63)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(64) = '1' ) then
llrduInfoLlrDelayLine_Next(63)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 64

if ( ph4DeMux(6) = '1' ) then
llrduInfoLlrDelayLine_Next(64)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(66) = '1' ) then
llrduInfoLlrDelayLine_Next(64)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(116) = '1' ) then
llrduInfoLlrDelayLine_Next(64)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(120) = '1' ) then
llrduInfoLlrDelayLine_Next(64)(2) <= llrParityTermOut5;

end if;
```

```
  -- Code for Info Registers 65

if ( ph4DeMux(39) = '1' ) then
llrduInfoLlrDelayLine_Next(65)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(74) = '1' ) then
llrduInfoLlrDelayLine_Next(65)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(92) = '1' ) then
llrduInfoLlrDelayLine_Next(65)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 66

if ( ph4DeMux(27) = '1' ) then
llrduInfoLlrDelayLine_Next(66)(1) <= llrParityTermOut4;

end if;
if ( ph4DeMux(2) = '1' ) then
llrduInfoLlrDelayLine_Next(66)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(117) = '1' ) then
llrduInfoLlrDelayLine_Next(66)(2) <= llrParityTermOut5;
end if;
if ( ph4DeMux(90) = '1' ) then
llrduInfoLlrDelayLine_Next(66)(3) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 67

if ( ph4DeMux(86) = '1' ) then
llrduInfoLlrDelayLine_Next(67)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(103) = '1' ) then
llrduInfoLlrDelayLine_Next(67)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(109) = '1' ) then
llrduInfoLlrDelayLine_Next(67)(2) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 68
if ( ph4DeMux(48) = '1' ) then
llrduInfoLlrDelayLine_Next(68)(2) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 69
if ( ph4DeMux(0) = '1' ) then
llrduInfoLlrDelayLine_Next(69)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(67) = '1' ) then
llrduInfoLlrDelayLine_Next(69)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(80) = '1' ) then
llrduInfoLlrDelayLine_Next(69)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(82) = '1' ) then
llrduInfoLlrDelayLine_Next(69)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(114) = '1' ) then
llrduInfoLlrDelayLine_Next(69)(2) <= llrParityTermOut4;
end if;
if ( ph4DeMux(99) = '1' ) then
llrduInfoLlrDelayLine_Next(69)(3) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 70
if ( ph4DeMux(19) = '1' ) then
llrduInfoLlrDelayLine_Next(70)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(73) = '1' ) then
llrduInfoLlrDelayLine_Next(70)(2) <= llrParityTermOut4;
elsif ( ph4DeMux(124) = '1' ) then
llrduInfoLlrDelayLine_Next(70)(2) <= llrParityTermOut5;
end if;
if ( ph4DeMux(87) = '1' ) then
llrduInfoLlrDelayLine_Next(70)(3) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 71
if ( ph4DeMux(81) = '1' ) then
llrduInfoLlrDelayLine_Next(71)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(31) = '1' ) then
llrduInfoLlrDelayLine_Next(71)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(110) = '1' ) then
llrduInfoLlrDelayLine_Next(71)(2) <= llrParityTermOut4;

end if;

  -- Code for Info Registers 72

if ( ph4DeMux(35) = '1' ) then
llrduInfoLlrDelayLine_Next(72)(1) <= llrParityTermOut5;

end if;
```

```
if ( ph4DeMux(12) = '1' ) then
llrduInfoLlrDelayLine_Next(72)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(25) = '1' ) then
llrduInfoLlrDelayLine_Next(72)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(53) = '1' ) then
llrduInfoLlrDelayLine_Next(72)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(79) = '1' ) then
llrduInfoLlrDelayLine_Next(72)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(98) = '1' ) then
llrduInfoLlrDelayLine_Next(72)(2) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 73

if ( ph4DeMux(32) = '1' ) then
llrduInfoLlrDelayLine_Next(73)(2) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 74

if ( ph4DeMux(7) = '1' ) then
llrduInfoLlrDelayLine_Next(74)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(42) = '1' ) then
llrduInfoLlrDelayLine_Next(74)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(45) = '1' ) then
llrduInfoLlrDelayLine_Next(74)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(88) = '1' ) then
llrduInfoLlrDelayLine_Next(74)(2) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 75

if ( ph4DeMux(100) = '1' ) then
llrduInfoLlrDelayLine_Next(75)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(104) = '1' ) then
llrduInfoLlrDelayLine_Next(75)(2) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 76

if ( ph4DeMux(55) = '1' ) then
llrduInfoLlrDelayLine_Next(76)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(70) = '1' ) then
llrduInfoLlrDelayLine_Next(76)(2) <= llrParityTermOut5;
end if;

  -- Code for Info Registers 77

if ( ph4DeMux(77) = '1' ) then
llrduInfoLlrDelayLine_Next(77)(2) <= llrParityTermOut5;

end if;
if ( ph4DeMux(3) = '1' ) then
llrduInfoLlrDelayLine_Next(77)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 78

if ( ph4DeMux(84) = '1' ) then
llrduInfoLlrDelayLine_Next(78)(1) <= llrParityTermOut5;

end if;
if ( ph4DeMux(40) = '1' ) then
llrduInfoLlrDelayLine_Next(78)(2) <= llrParityTermOut5;

end if;
if ( ph4DeMux(10) = '1' ) then
llrduInfoLlrDelayLine_Next(78)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 79

if ( ph4DeMux(62) = '1' ) then
llrduInfoLlrDelayLine_Next(79)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(119) = '1' ) then
llrduInfoLlrDelayLine_Next(79)(2) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 80

if ( ph4DeMux(36) = '1' ) then
llrduInfoLlrDelayLine_Next(80)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(128) = '1' ) then
llrduInfoLlrDelayLine_Next(80)(2) <= llrParityTermOut5;

end if;
```

```
  -- Code for Info Registers 81

if ( ph4DeMux(29) = '1' ) then
llrduInfoLlrDelayLine_Next(81)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(38) = '1' ) then
llrduInfoLlrDelayLine_Next(81)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(63) = '1' ) then
llrduInfoLlrDelayLine_Next(81)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(115) = '1' ) then
llrduInfoLlrDelayLine_Next(81)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(5) = '1' ) then
llrduInfoLlrDelayLine_Next(81)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(22) = '1' ) then
llrduInfoLlrDelayLine_Next(81)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 82

if ( ph4DeMux(17) = '1' ) then
llrduInfoLlrDelayLine_Next(82)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(71) = '1' ) then
llrduInfoLlrDelayLine_Next(82)(2) <= llrParityTermOut5;

end if;
if ( ph4DeMux(20) = '1' ) then
llrduInfoLlrDelayLine_Next(82)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(89) = '1' ) then
llrduInfoLlrDelayLine_Next(82)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 83

if ( ph4DeMux(8) = '1' ) then
llrduInfoLlrDelayLine_Next(83)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(114) = '1' ) then
llrduInfoLlrDelayLine_Next(83)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(118) = '1' ) then
llrduInfoLlrDelayLine_Next(83)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 84

if ( ph4DeMux(35) = '1' ) then
llrduInfoLlrDelayLine_Next(84)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(77) = '1' ) then
llrduInfoLlrDelayLine_Next(84)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(94) = '1' ) then
llrduInfoLlrDelayLine_Next(84)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(46) = '1' ) then
llrduInfoLlrDelayLine_Next(84)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 85

if ( ph4DeMux(13) = '1' ) then
llrduInfoLlrDelayLine_Next(85)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(101) = '1' ) then
llrduInfoLlrDelayLine_Next(85)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(122) = '1' ) then
llrduInfoLlrDelayLine_Next(85)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(54) = '1' ) then
llrduInfoLlrDelayLine_Next(85)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 86

if ( ph4DeMux(106) = '1' ) then
llrduInfoLlrDelayLine_Next(86)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(56) = '1' ) then
llrduInfoLlrDelayLine_Next(86)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 87
```

```
if ( ph4DeMux(119) = '1' ) then
llrduInfoLlrDelayLine_Next(87)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(120) = '1' ) then
llrduInfoLlrDelayLine_Next(87)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(91) = '1' ) then
llrduInfoLlrDelayLine_Next(87)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 88

if ( ph4DeMux(75) = '1' ) then
llrduInfoLlrDelayLine_Next(88)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(85) = '1' ) then
llrduInfoLlrDelayLine_Next(88)(2) <= llrParityTermOut5;

end if;
if ( ph4DeMux(52) = '1' ) then
llrduInfoLlrDelayLine_Next(88)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 89

if ( ph4DeMux(41) = '1' ) then
llrduInfoLlrDelayLine_Next(89)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(97) = '1' ) then
llrduInfoLlrDelayLine_Next(89)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(127) = '1' ) then
llrduInfoLlrDelayLine_Next(89)(2) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 90

if ( ph4DeMux(27) = '1' ) then
llrduInfoLlrDelayLine_Next(90)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(37) = '1' ) then
llrduInfoLlrDelayLine_Next(90)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(74) = '1' ) then
llrduInfoLlrDelayLine_Next(90)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 91


if ( ph4DeMux(25) = '1' ) then
llrduInfoLlrDelayLine_Next(91)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(93) = '1' ) then
llrduInfoLlrDelayLine_Next(91)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 92

if ( ph4DeMux(31) = '1' ) then
llrduInfoLlrDelayLine_Next(92)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(38) = '1' ) then
llrduInfoLlrDelayLine_Next(92)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(14) = '1' ) then
llrduInfoLlrDelayLine_Next(92)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(59) = '1' ) then
llrduInfoLlrDelayLine_Next(92)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(95) = '1' ) then
llrduInfoLlrDelayLine_Next(92)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 93

if ( ph4DeMux(11) = '1' ) then
llrduInfoLlrDelayLine_Next(93)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(67) = '1' ) then
llrduInfoLlrDelayLine_Next(93)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(111) = '1' ) then
llrduInfoLlrDelayLine_Next(93)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(116) = '1' ) then
llrduInfoLlrDelayLine_Next(93)(2) <= llrParityTermOut5;

end if;
if ( ph4DeMux(61) = '1' ) then
llrduInfoLlrDelayLine_Next(93)(3) <= llrParityTermOut5;
```

```
end if;

  -- Code for Info Registers 94

if ( ph4DeMux(109) = '1' ) then
llrduInfoLlrDelayLine_Next(94)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(21) = '1' ) then
llrduInfoLlrDelayLine_Next(94)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 95

if ( ph4DeMux(40) = '1' ) then
llrduInfoLlrDelayLine_Next(95)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(96) = '1' ) then
llrduInfoLlrDelayLine_Next(95)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 96

if ( ph4DeMux(45) = '1' ) then
llrduInfoLlrDelayLine_Next(96)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(105) = '1' ) then
llrduInfoLlrDelayLine_Next(96)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 97

if ( ph4DeMux(26) = '1' ) then
llrduInfoLlrDelayLine_Next(97)(2) <= llrParityTermOut5;
elsif ( ph4DeMux(102) = '1' ) then
llrduInfoLlrDelayLine_Next(97)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(57) = '1' ) then
llrduInfoLlrDelayLine_Next(97)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(92) = '1' ) then
llrduInfoLlrDelayLine_Next(97)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(108) = '1' ) then
llrduInfoLlrDelayLine_Next(97)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 98

if ( ph4DeMux(28) = '1' ) then
llrduInfoLlrDelayLine_Next(98)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(83) = '1' ) then
llrduInfoLlrDelayLine_Next(98)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(126) = '1' ) then
llrduInfoLlrDelayLine_Next(98)(2) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 99

if ( ph4DeMux(8) = '1' ) then
llrduInfoLlrDelayLine_Next(99)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(15) = '1' ) then
llrduInfoLlrDelayLine_Next(99)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 100

if ( ph4DeMux(91) = '1' ) then
llrduInfoLlrDelayLine_Next(100)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 101

if ( ph4DeMux(18) = '1' ) then
llrduInfoLlrDelayLine_Next(101)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(58) = '1' ) then
llrduInfoLlrDelayLine_Next(101)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(87) = '1' ) then
llrduInfoLlrDelayLine_Next(101)(3) <= llrParityTermOut6;

end if;
```

```
  -- Code for Info Registers 102

if ( ph4DeMux(65) = '1' ) then
llrduInfoLlrDelayLine_Next(102)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(80) = '1' ) then
llrduInfoLlrDelayLine_Next(102)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(47) = '1' ) then
llrduInfoLlrDelayLine_Next(102)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(48) = '1' ) then
llrduInfoLlrDelayLine_Next(102)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 103

if ( ph4DeMux(64) = '1' ) then
llrduInfoLlrDelayLine_Next(103)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(125) = '1' ) then
llrduInfoLlrDelayLine_Next(103)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(16) = '1' ) then
llrduInfoLlrDelayLine_Next(103)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(76) = '1' ) then
llrduInfoLlrDelayLine_Next(103)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 104

if ( ph4DeMux(23) = '1' ) then
llrduInfoLlrDelayLine_Next(104)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(52) = '1' ) then
llrduInfoLlrDelayLine_Next(104)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(69) = '1' ) then
llrduInfoLlrDelayLine_Next(104)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(79) = '1' ) then
llrduInfoLlrDelayLine_Next(104)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 105

if ( ph4DeMux(81) = '1' ) then
llrduInfoLlrDelayLine_Next(105)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(117) = '1' ) then
llrduInfoLlrDelayLine_Next(105)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 106


  -- Code for Info Registers 107

if ( ph4DeMux(113) = '1' ) then
llrduInfoLlrDelayLine_Next(107)(2) <= llrParityTermOut6;
elsif ( ph4DeMux(128) = '1' ) then
llrduInfoLlrDelayLine_Next(107)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(3) = '1' ) then
llrduInfoLlrDelayLine_Next(107)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(60) = '1' ) then
llrduInfoLlrDelayLine_Next(107)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(73) = '1' ) then
llrduInfoLlrDelayLine_Next(107)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(84) = '1' ) then
llrduInfoLlrDelayLine_Next(107)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(95) = '1' ) then
llrduInfoLlrDelayLine_Next(107)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(97) = '1' ) then
llrduInfoLlrDelayLine_Next(107)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 108

if ( ph4DeMux(107) = '1' ) then
llrduInfoLlrDelayLine_Next(108)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 109

if ( ph4DeMux(0) = '1' ) then
llrduInfoLlrDelayLine_Next(109)(3) <= llrParityTermOut6;
```

```
elsif ( ph4DeMux(30) = '1' ) then
llrduInfoLlrDelayLine_Next(109)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(103) = '1' ) then
llrduInfoLlrDelayLine_Next(109)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 110

if ( ph4DeMux(24) = '1' ) then
llrduInfoLlrDelayLine_Next(110)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(66) = '1' ) then
llrduInfoLlrDelayLine_Next(110)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(82) = '1' ) then
llrduInfoLlrDelayLine_Next(110)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(99) = '1' ) then
llrduInfoLlrDelayLine_Next(110)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(110) = '1' ) then
llrduInfoLlrDelayLine_Next(110)(3) <= llrParityTermOut5;

end if;

  -- Code for Info Registers 111

if ( ph4DeMux(61) = '1' ) then
llrduInfoLlrDelayLine_Next(111)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(90) = '1' ) then
llrduInfoLlrDelayLine_Next(111)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(124) = '1' ) then
llrduInfoLlrDelayLine_Next(111)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 112

if ( ph4DeMux(2) = '1' ) then
llrduInfoLlrDelayLine_Next(112)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 113

if ( ph4DeMux(17) = '1' ) then
llrduInfoLlrDelayLine_Next(113)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(36) = '1' ) then
llrduInfoLlrDelayLine_Next(113)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(68) = '1' ) then
llrduInfoLlrDelayLine_Next(113)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(72) = '1' ) then
llrduInfoLlrDelayLine_Next(113)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(93) = '1' ) then
llrduInfoLlrDelayLine_Next(113)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 114

if ( ph4DeMux(19) = '1' ) then
llrduInfoLlrDelayLine_Next(114)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(51) = '1' ) then
llrduInfoLlrDelayLine_Next(114)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(96) = '1' ) then
llrduInfoLlrDelayLine_Next(114)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 115

if ( ph4DeMux(14) = '1' ) then
llrduInfoLlrDelayLine_Next(115)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(43) = '1' ) then
llrduInfoLlrDelayLine_Next(115)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 116

if ( ph4DeMux(63) = '1' ) then
llrduInfoLlrDelayLine_Next(116)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(112) = '1' ) then
llrduInfoLlrDelayLine_Next(116)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 117

if ( ph4DeMux(4) = '1' ) then
llrduInfoLlrDelayLine_Next(117)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(88) = '1' ) then
```

```
llrduInfoLlrDelayLine_Next(117)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 118

if ( ph4DeMux(33) = '1' ) then
llrduInfoLlrDelayLine_Next(118)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(49) = '1' ) then
llrduInfoLlrDelayLine_Next(118)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(58) = '1' ) then
llrduInfoLlrDelayLine_Next(118)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 119

if ( ph4DeMux(5) = '1' ) then
llrduInfoLlrDelayLine_Next(119)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(27) = '1' ) then
llrduInfoLlrDelayLine_Next(119)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(70) = '1' ) then
llrduInfoLlrDelayLine_Next(119)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 120

if ( ph4DeMux(12) = '1' ) then
llrduInfoLlrDelayLine_Next(120)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(53) = '1' ) then
llrduInfoLlrDelayLine_Next(120)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(72) = '1' ) then
llrduInfoLlrDelayLine_Next(120)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(78) = '1' ) then
llrduInfoLlrDelayLine_Next(120)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(105) = '1' ) then
llrduInfoLlrDelayLine_Next(120)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 121

if ( ph4DeMux(32) = '1' ) then
llrduInfoLlrDelayLine_Next(121)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(57) = '1' ) then
llrduInfoLlrDelayLine_Next(121)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(114) = '1' ) then
llrduInfoLlrDelayLine_Next(121)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 122

if ( ph4DeMux(42) = '1' ) then
llrduInfoLlrDelayLine_Next(122)(2) <= llrParityTermOut6;

end if;
if ( ph4DeMux(66) = '1' ) then
llrduInfoLlrDelayLine_Next(122)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 123

if ( ph4DeMux(16) = '1' ) then
llrduInfoLlrDelayLine_Next(123)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(86) = '1' ) then
llrduInfoLlrDelayLine_Next(123)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(101) = '1' ) then
llrduInfoLlrDelayLine_Next(123)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(104) = '1' ) then
llrduInfoLlrDelayLine_Next(123)(3) <= llrParityTermOut6;

end if;

  -- Code for Info Registers 124

if ( ph4DeMux(22) = '1' ) then
llrduInfoLlrDelayLine_Next(124)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(24) = '1' ) then
llrduInfoLlrDelayLine_Next(124)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(34) = '1' ) then
llrduInfoLlrDelayLine_Next(124)(3) <= llrParityTermOut5;
elsif ( ph4DeMux(98) = '1' ) then
llrduInfoLlrDelayLine_Next(124)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(121) = '1' ) then
llrduInfoLlrDelayLine_Next(124)(3) <= llrParityTermOut5;
end if;
```

```
  -- Code for Info Registers 125
if ( ph4DeMux(1) = '1' ) then
llrduInfoLlrDelayLine_Next(125)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(54) = '1' ) then
llrduInfoLlrDelayLine_Next(125)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(123) = '1' ) then
llrduInfoLlrDelayLine_Next(125)(3) <= llrParityTermOut6;
end if;

  -- Code for Info Registers 126
if ( ph4DeMux(44) = '1' ) then
llrduInfoLlrDelayLine_Next(126)(3) <= llrParityTermOut6;
end if;

  -- Code for Info Registers 127
if ( ph4DeMux(30) = '1' ) then
llrduInfoLlrDelayLine_Next(127)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(34) = '1' ) then
llrduInfoLlrDelayLine_Next(127)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(39) = '1' ) then
llrduInfoLlrDelayLine_Next(127)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(62) = '1' ) then
llrduInfoLlrDelayLine_Next(127)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(110) = '1' ) then
llrduInfoLlrDelayLine_Next(127)(3) <= llrParityTermOut6;
end if;

  -- Code for Info Registers 128
if ( ph4DeMux(67) = '1' ) then
llrduInfoLlrDelayLine_Next(128)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(71) = '1' ) then
llrduInfoLlrDelayLine_Next(128)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(89) = '1' ) then
llrduInfoLlrDelayLine_Next(128)(3) <= llrParityTermOut6;
end if;

  -- Code for Info Registers 129
if ( ph4DeMux(6) = '1' ) then
llrduInfoLlrDelayLine_Next(129)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(10) = '1' ) then
llrduInfoLlrDelayLine_Next(129)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(18) = '1' ) then
llrduInfoLlrDelayLine_Next(129)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(23) = '1' ) then
llrduInfoLlrDelayLine_Next(129)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(26) = '1' ) then
llrduInfoLlrDelayLine_Next(129)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(59) = '1' ) then
llrduInfoLlrDelayLine_Next(129)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(116) = '1' ) then
llrduInfoLlrDelayLine_Next(129)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(121) = '1' ) then
llrduInfoLlrDelayLine_Next(129)(3) <= llrParityTermOut6;
end if;

  -- Code for Info Registers 130
if ( ph4DeMux(9) = '1' ) then
llrduInfoLlrDelayLine_Next(130)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(15) = '1' ) then
llrduInfoLlrDelayLine_Next(130)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(50) = '1' ) then
llrduInfoLlrDelayLine_Next(130)(3) <= llrParityTermOut6;
end if;

  -- Code for Info Registers 131
if ( ph4DeMux(73) = '1' ) then
llrduInfoLlrDelayLine_Next(131)(3) <= llrParityTermOut6;
elsif ( ph4DeMux(85) = '1' ) then
llrduInfoLlrDelayLine_Next(131)(3) <= llrParityTermOut6;
end if;

  end process;

    -- elaborate time of this process by itself: ~20min
    process ( llrduCodeLlrDelayLine, ph4DeMux,
        llrParityTermOut1 , llrParityTermOut2 , llrParityTermOut3 ) is

   begin

llrduCodeLlrDelayLine_Next(139 downto 1) <= llrduCodeLlrDelayLine(138 downto 0);

  -- Code for Code Registers 13
if ( ph4DeMux(1) = '1' ) then
llrduCodeLlrDelayLine_Next(13)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(89) = '1' ) then
llrduCodeLlrDelayLine_Next(13)(2) <= llrParityTermOut3;
end if;
```

```
  -- Code for Code Registers 15
if ( ph4DeMux(0) = '1' ) then
llrduCodeLlrDelayLine_Next(15)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(45) = '1' ) then
llrduCodeLlrDelayLine_Next(15)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(46) = '1' ) then
llrduCodeLlrDelayLine_Next(15)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(96) = '1' ) then
llrduCodeLlrDelayLine_Next(15)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 17
if ( ph4DeMux(92) = '1' ) then
llrduCodeLlrDelayLine_Next(17)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(95) = '1' ) then
llrduCodeLlrDelayLine_Next(17)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 18
if ( ph4DeMux(50) = '1' ) then
llrduCodeLlrDelayLine_Next(18)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 19
if ( ph4DeMux(11) = '1' ) then
llrduCodeLlrDelayLine_Next(19)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(14) = '1' ) then
llrduCodeLlrDelayLine_Next(19)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(48) = '1' ) then
llrduCodeLlrDelayLine_Next(19)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 22
if ( ph4DeMux(86) = '1' ) then
llrduCodeLlrDelayLine_Next(22)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 24
if ( ph4DeMux(3) = '1' ) then
llrduCodeLlrDelayLine_Next(24)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(62) = '1' ) then
llrduCodeLlrDelayLine_Next(24)(2) <= llrParityTermOut3;
else
llrduCodeLlrDelayLine_Next(24)(2) <= llrduCodeLlrDelayLine(23)(2);
end if;

  -- Code for Code Registers 25
if ( ph4DeMux(18) = '1' ) then
llrduCodeLlrDelayLine_Next(25)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 26
if ( ph4DeMux(6) = '1' ) then
llrduCodeLlrDelayLine_Next(26)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(13) = '1' ) then
llrduCodeLlrDelayLine_Next(26)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 27
if ( ph4DeMux(87) = '1' ) then
llrduCodeLlrDelayLine_Next(27)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 29
if ( ph4DeMux(12) = '1' ) then
llrduCodeLlrDelayLine_Next(29)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(103) = '1' ) then
llrduCodeLlrDelayLine_Next(29)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 31
if ( ph4DeMux(43) = '1' ) then
llrduCodeLlrDelayLine_Next(31)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 32
if ( ph4DeMux(5) = '1' ) then
llrduCodeLlrDelayLine_Next(32)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(55) = '1' ) then
llrduCodeLlrDelayLine_Next(32)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 33
if ( ph4DeMux(2) = '1' ) then
llrduCodeLlrDelayLine_Next(33)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(100) = '1' ) then
llrduCodeLlrDelayLine_Next(33)(2) <= llrParityTermOut3;
end if;
```

```
  -- Code for Code Registers 34
if ( ph4DeMux(70) = '1' ) then
llrduCodeLlrDelayLine_Next(34)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(105) = '1' ) then
llrduCodeLlrDelayLine_Next(34)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 35
if ( ph4DeMux(7) = '1' ) then
llrduCodeLlrDelayLine_Next(35)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(16) = '1' ) then
llrduCodeLlrDelayLine_Next(35)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(94) = '1' ) then
llrduCodeLlrDelayLine_Next(35)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(101) = '1' ) then
llrduCodeLlrDelayLine_Next(35)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 36
if ( ph4DeMux(49) = '1' ) then
llrduCodeLlrDelayLine_Next(36)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 37
if ( ph4DeMux(54) = '1' ) then
llrduCodeLlrDelayLine_Next(37)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(58) = '1' ) then
llrduCodeLlrDelayLine_Next(37)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(88) = '1' ) then
llrduCodeLlrDelayLine_Next(37)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(107) = '1' ) then
llrduCodeLlrDelayLine_Next(37)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 38
if ( ph4DeMux(8) = '1' ) then
llrduCodeLlrDelayLine_Next(38)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(56) = '1' ) then
llrduCodeLlrDelayLine_Next(38)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(71) = '1' ) then
llrduCodeLlrDelayLine_Next(38)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 39
if ( ph4DeMux(17) = '1' ) then
llrduCodeLlrDelayLine_Next(39)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(59) = '1' ) then
llrduCodeLlrDelayLine_Next(39)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(76) = '1' ) then
llrduCodeLlrDelayLine_Next(39)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(108) = '1' ) then
llrduCodeLlrDelayLine_Next(39)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(111) = '1' ) then
llrduCodeLlrDelayLine_Next(39)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 40
if ( ph4DeMux(24) = '1' ) then
llrduCodeLlrDelayLine_Next(40)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(65) = '1' ) then
llrduCodeLlrDelayLine_Next(40)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 41
if ( ph4DeMux(44) = '1' ) then
llrduCodeLlrDelayLine_Next(41)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(121) = '1' ) then
llrduCodeLlrDelayLine_Next(41)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 42
if ( ph4DeMux(4) = '1' ) then
llrduCodeLlrDelayLine_Next(42)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 43
if ( ph4DeMux(9) = '1' ) then
llrduCodeLlrDelayLine_Next(43)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(20) = '1' ) then
llrduCodeLlrDelayLine_Next(43)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 44
if ( ph4DeMux(68) = '1' ) then
llrduCodeLlrDelayLine_Next(44)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(91) = '1' ) then
llrduCodeLlrDelayLine_Next(44)(2) <= llrParityTermOut3;
end if;
```

```
  -- Code for Code Registers 45
if ( ph4DeMux(47) = '1' ) then
llrduCodeLlrDelayLine_Next(45)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(67) = '1' ) then
llrduCodeLlrDelayLine_Next(45)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(99) = '1' ) then
llrduCodeLlrDelayLine_Next(45)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 46
if ( ph4DeMux(10) = '1' ) then
llrduCodeLlrDelayLine_Next(46)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(73) = '1' ) then
llrduCodeLlrDelayLine_Next(46)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(74) = '1' ) then
llrduCodeLlrDelayLine_Next(46)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(123) = '1' ) then
llrduCodeLlrDelayLine_Next(46)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 47
if ( ph4DeMux(21) = '1' ) then
llrduCodeLlrDelayLine_Next(47)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(36) = '1' ) then
llrduCodeLlrDelayLine_Next(47)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(90) = '1' ) then
llrduCodeLlrDelayLine_Next(47)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(93) = '1' ) then
llrduCodeLlrDelayLine_Next(47)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(110) = '1' ) then
llrduCodeLlrDelayLine_Next(47)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(126) = '1' ) then
llrduCodeLlrDelayLine_Next(47)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 48
if ( ph4DeMux(23) = '1' ) then
llrduCodeLlrDelayLine_Next(48)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(34) = '1' ) then
llrduCodeLlrDelayLine_Next(48)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(38) = '1' ) then
llrduCodeLlrDelayLine_Next(48)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(42) = '1' ) then
llrduCodeLlrDelayLine_Next(48)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(64) = '1' ) then
llrduCodeLlrDelayLine_Next(48)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(82) = '1' ) then
llrduCodeLlrDelayLine_Next(48)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(97) = '1' ) then
llrduCodeLlrDelayLine_Next(48)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(113) = '1' ) then
llrduCodeLlrDelayLine_Next(48)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 49
if ( ph4DeMux(84) = '1' ) then
llrduCodeLlrDelayLine_Next(49)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 50
if ( ph4DeMux(41) = '1' ) then
llrduCodeLlrDelayLine_Next(50)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(98) = '1' ) then
llrduCodeLlrDelayLine_Next(50)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(118) = '1' ) then
llrduCodeLlrDelayLine_Next(50)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 51
if ( ph4DeMux(52) = '1' ) then
llrduCodeLlrDelayLine_Next(51)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(124) = '1' ) then
llrduCodeLlrDelayLine_Next(51)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 52
if ( ph4DeMux(51) = '1' ) then
llrduCodeLlrDelayLine_Next(52)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(78) = '1' ) then
llrduCodeLlrDelayLine_Next(52)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 53
if ( ph4DeMux(29) = '1' ) then
llrduCodeLlrDelayLine_Next(53)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(61) = '1' ) then
llrduCodeLlrDelayLine_Next(53)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(63) = '1' ) then
llrduCodeLlrDelayLine_Next(53)(2) <= llrParityTermOut3;
```

```
elsif ( ph4DeMux(106) = '1' ) then
llrduCodeLlrDelayLine_Next(53)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 54
if ( ph4DeMux(109) = '1' ) then
llrduCodeLlrDelayLine_Next(54)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 55
if ( ph4DeMux(15) = '1' ) then
llrduCodeLlrDelayLine_Next(55)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(60) = '1' ) then
llrduCodeLlrDelayLine_Next(55)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(112) = '1' ) then
llrduCodeLlrDelayLine_Next(55)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 56
if ( ph4DeMux(114) = '1' ) then
llrduCodeLlrDelayLine_Next(56)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(117) = '1' ) then
llrduCodeLlrDelayLine_Next(56)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 57
if ( ph4DeMux(25) = '1' ) then
llrduCodeLlrDelayLine_Next(57)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(39) = '1' ) then
llrduCodeLlrDelayLine_Next(57)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(53) = '1' ) then
llrduCodeLlrDelayLine_Next(57)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(57) = '1' ) then
llrduCodeLlrDelayLine_Next(57)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 58
if ( ph4DeMux(69) = '1' ) then
llrduCodeLlrDelayLine_Next(58)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(120) = '1' ) then
llrduCodeLlrDelayLine_Next(58)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 59
if ( ph4DeMux(115) = '1' ) then
llrduCodeLlrDelayLine_Next(59)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 60
if ( ph4DeMux(102) = '1' ) then
llrduCodeLlrDelayLine_Next(60)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 62
if ( ph4DeMux(19) = '1' ) then
llrduCodeLlrDelayLine_Next(62)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(66) = '1' ) then
llrduCodeLlrDelayLine_Next(62)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(77) = '1' ) then
llrduCodeLlrDelayLine_Next(62)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 64
if ( ph4DeMux(35) = '1' ) then
llrduCodeLlrDelayLine_Next(64)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(83) = '1' ) then
llrduCodeLlrDelayLine_Next(64)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(116) = '1' ) then
llrduCodeLlrDelayLine_Next(64)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 65
if ( ph4DeMux(32) = '1' ) then
llrduCodeLlrDelayLine_Next(65)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(104) = '1' ) then
llrduCodeLlrDelayLine_Next(65)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 66
if ( ph4DeMux(27) = '1' ) then
llrduCodeLlrDelayLine_Next(66)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(80) = '1' ) then
llrduCodeLlrDelayLine_Next(66)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 67
if ( ph4DeMux(26) = '1' ) then
llrduCodeLlrDelayLine_Next(67)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(30) = '1' ) then
```

```
llrduCodeLlrDelayLine_Next(67)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 68
if ( ph4DeMux(22) = '1' ) then
llrduCodeLlrDelayLine_Next(68)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 72
if ( ph4DeMux(28) = '1' ) then
llrduCodeLlrDelayLine_Next(72)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 73
if ( ph4DeMux(79) = '1' ) then
llrduCodeLlrDelayLine_Next(73)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 74
if ( ph4DeMux(81) = '1' ) then
llrduCodeLlrDelayLine_Next(74)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 75
if ( ph4DeMux(40) = '1' ) then
llrduCodeLlrDelayLine_Next(75)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(72) = '1' ) then
llrduCodeLlrDelayLine_Next(75)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(125) = '1' ) then
llrduCodeLlrDelayLine_Next(75)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 76
if ( ph4DeMux(31) = '1' ) then
llrduCodeLlrDelayLine_Next(76)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(85) = '1' ) then
llrduCodeLlrDelayLine_Next(76)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 77
if ( ph4DeMux(75) = '1' ) then
llrduCodeLlrDelayLine_Next(77)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 78
if ( ph4DeMux(119) = '1' ) then
llrduCodeLlrDelayLine_Next(78)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 79
if ( ph4DeMux(37) = '1' ) then
llrduCodeLlrDelayLine_Next(79)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 80
if ( ph4DeMux(33) = '1' ) then
llrduCodeLlrDelayLine_Next(80)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 82
if ( ph4DeMux(122) = '1' ) then
llrduCodeLlrDelayLine_Next(82)(2) <= llrParityTermOut3;
elsif ( ph4DeMux(127) = '1' ) then
llrduCodeLlrDelayLine_Next(82)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 84
if ( ph4DeMux(128) = '1' ) then
llrduCodeLlrDelayLine_Next(84)(2) <= llrParityTermOut3;
end if;

  -- Code for Code Registers 132
llrduCodeLlrDelayLine_Next(132)(3) <= llrParityTermOut2;

-- End of auto-gen code
   end process;

end Behavioral;
```

Listing B.10: LdpccParityCheckNodeWithInf.vhd

```
--
--    Filename: LdpccParityCheckNodeWithInf.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
```

```
--
--    Author:    Stephen Bates & Ramkrishna Swamy
--    Location:  Dept. of Electrical and Computer Engineering
--               The University of Alberta
--               Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------
--
--   Description
--   -----------
--
--  This is a 6 input 6 output parity check node for LDPC-CCs which can operate
--  with infinite inputs.
--
-- ------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity LdpccParityCheckNodeWithInf is
    Port ( clkClock        : in LogicBit;
           rstReset        : in LogicBit;
           llrLlrTermIn1   : in LogLikeRatio;
           llrLlrTermIn2   : in LogLikeRatio;
           llrLlrTermIn3   : in LogLikeRatio;
           llrLlrTermIn4   : in LogLikeRatio;
           llrLlrTermIn5   : in LogLikeRatio;
           llrLlrTermIn6   : in LogLikeRatio;
           --Outputs
           llrLlrTermOut1  : out LogLikeRatio;
           llrLlrTermOut2  : out LogLikeRatio;
           llrLlrTermOut3  : out LogLikeRatio;
           llrLlrTermOut4  : out LogLikeRatio;
           llrLlrTermOut5  : out LogLikeRatio;
           llrLlrTermOut6  : out LogLikeRatio
         );
end LdpccParityCheckNodeWithInf;

architecture Behavioral of LdpccParityCheckNodeWithInf is

-- ------------------------------------------------------------------
-- COMPONENTS
-- ------------------------------------------------------------------

    -- Components which we instantiate within the main
    -- block.

   component LdpccMinMaxComp
    port ( mmtMinMaxCompIn1     : in MinMaxTerm;
           mmtMinMaxCompIn2     : in MinMaxTerm;
           --Outputs
           mmtMinMaxCompMinOut  : out MinMaxTerm;
           mmtMinMaxCompMaxOut  : out MinMaxTerm
         );
    end component;

-- ------------------------------------------------------------------
-- REGISTERS
-- ------------------------------------------------------------------

    -- Registered values which must have an _Next value
    -- to work well. As per Massana design rules.

   signal mmtMinimum_Next          : MinMaxTerm;
   signal mmtMinimum               : MinMaxTerm;

   signal mmtSecondMinimum_Next    : MinMaxTerm;
   signal mmtSecondMinimum         : MinMaxTerm;

   signal llrLlrTermOut1Reg_Next : LogLikeRatio;
   signal llrLlrTermOut1Reg      : LogLikeRatio;
   signal llrLlrTermOut2Reg_Next : LogLikeRatio;
   signal llrLlrTermOut2Reg      : LogLikeRatio;
   signal llrLlrTermOut3Reg_Next : LogLikeRatio;
   signal llrLlrTermOut3Reg      : LogLikeRatio;
   signal llrLlrTermOut4Reg_Next : LogLikeRatio;
   signal llrLlrTermOut4Reg      : LogLikeRatio;
   signal llrLlrTermOut5Reg_Next : LogLikeRatio;
   signal llrLlrTermOut5Reg      : LogLikeRatio;
   signal llrLlrTermOut6Reg_Next : LogLikeRatio;
   signal llrLlrTermOut6Reg      : LogLikeRatio;

   signal lbSignBitTerm1_Del1_Next : LogicBit;
   signal lbSignBitTerm1_Del1      : LogicBit;
   signal lbSignBitTerm2_Del1_Next : LogicBit;
```

```
signal lbSignBitTerm2_Del1       : LogicBit;
signal lbSignBitTerm3_Del1_Next  : LogicBit;
signal lbSignBitTerm3_Del1       : LogicBit;
signal lbSignBitTerm4_Del1_Next  : LogicBit;
signal lbSignBitTerm4_Del1       : LogicBit;
signal lbSignBitTerm5_Del1_Next  : LogicBit;
signal lbSignBitTerm5_Del1       : LogicBit;
signal lbSignBitTerm6_Del1_Next  : LogicBit;
signal lbSignBitTerm6_Del1       : LogicBit;

signal lbTotalSign_Next : LogicBit;
signal lbTotalSign      : LogicBit;

-- The following signals are defined to eliminate the
-- WARNINGS during synthesis
signal llrLlrTermIn1_Sgn_tmp : LogicBit;
signal llrLlrTermIn2_Sgn_tmp : LogicBit;
signal llrLlrTermIn3_Sgn_tmp : LogicBit;
signal llrLlrTermIn4_Sgn_tmp : LogicBit;
signal llrLlrTermIn5_Sgn_tmp : LogicBit;
signal llrLlrTermIn6_Sgn_tmp : LogicBit;

-------------------------------------------------------------------------------
-- COMBINATORIAL
-------------------------------------------------------------------------------

  -- Combinatorial signals which are not assigned to registers.

signal mmtMinMaxTermIn1  : MinMaxTerm;
signal mmtMinMaxTermIn2  : MinMaxTerm;
signal mmtMinMaxTermIn3  : MinMaxTerm;
signal mmtMinMaxTermIn4  : MinMaxTerm;
signal mmtMinMaxTermIn5  : MinMaxTerm;
signal mmtMinMaxTermIn6  : MinMaxTerm;

signal mmtMinStageOneIdOne    : MinMaxTerm;
signal mmtMaxStageOneIdOne    : MinMaxTerm;
signal mmtMinStageOneIdTwo    : MinMaxTerm;
signal mmtMaxStageOneIdTwo    : MinMaxTerm;
signal mmtMinStageOneIdThree  : MinMaxTerm;
signal mmtMaxStageOneIdThree  : MinMaxTerm;

signal mmtMinStageTwoMinIdOne : MinMaxTerm;
signal mmtMaxStageTwoMinIdOne : MinMaxTerm;
signal mmtMaxStageTwoMinIdTwo : MinMaxTerm;

signal mmtMinStageTwoMaxIdOne : MinMaxTerm;
signal mmtMinStageTwoMaxIdTwo : MinMaxTerm;

signal mmtMinStageThreeMaxIdOne : MinMaxTerm;

-------------------------------------------------------------------------------
-- PROCEDURES and FUNCTIONS
-------------------------------------------------------------------------------

  -- Procedures and functions we might use to speed up code.

begin

-------------------------------------------------------------------------------
-- INSTANTIATIONS
-------------------------------------------------------------------------------

  -- Instantiate the max/min entities which determine both the
  -- minima and 2nd minima of the 6 inputs. We do this is a cascaded
  -- fashion. Not these entities are combinatorial and we add
  -- pipelining externally if required.

 MinMaxCompStageOneIdOne : LdpccMinMaxComp
 port map ( mmtMinMaxCompIn1      => mmtMinMaxTermIn1,
            mmtMinMaxCompIn2      => mmtMinMaxTermIn2,
            --Outputs
            mmtMinMaxCompMinOut => mmtMinStageOneIdOne,
            mmtMinMaxCompMaxOut => mmtMaxStageOneIdOne
          );

MinMaxCompStageOneIdTwo : LdpccMinMaxComp
 port map ( mmtMinMaxCompIn1      => mmtMinMaxTermIn3,
            mmtMinMaxCompIn2      => mmtMinMaxTermIn4,
            --Outputs
            mmtMinMaxCompMinOut => mmtMinStageOneIdTwo,
            mmtMinMaxCompMaxOut => mmtMaxStageOneIdTwo
          );

MinMaxCompStageOneIdThree : LdpccMinMaxComp
 port map ( mmtMinMaxCompIn1      => mmtMinMaxTermIn5,
            mmtMinMaxCompIn2      => mmtMinMaxTermIn6,
            --Outputs
            mmtMinMaxCompMinOut => mmtMinStageOneIdThree,
```

```
            mmtMinMaxCompMaxOut => mmtMaxStageOneIdThree
           );

MinMaxCompStageTwoMinIdOne : LdpccMinMaxComp
 port map ( mmtMinMaxCompIn1    => mmtMinStageOneIdOne,
            mmtMinMaxCompIn2    => mmtMinStageOneIdTwo,
            --Outputs
            mmtMinMaxCompMinOut => mmtMinStageTwoMinIdOne,
            mmtMinMaxCompMaxOut => mmtMaxStageTwoMinIdOne
           );

MinMaxCompStageTwoMinIdTwo : LdpccMinMaxComp
 port map ( mmtMinMaxCompIn1    => mmtMinStageOneIdThree,
            mmtMinMaxCompIn2    => mmtMinStageTwoMinIdOne,
            --Outputs
            mmtMinMaxCompMinOut => mmtMinimum_Next,
            mmtMinMaxCompMaxOut => mmtMaxStageTwoMinIdTwo
           );

MinMaxCompStageTwoMaxIdOne : LdpccMinMaxComp
port map ( mmtMinMaxCompIn1    => mmtMaxStageOneIdOne,
            mmtMinMaxCompIn2    => mmtMaxStageOneIdTwo,
            --Outputs
            mmtMinMaxCompMinOut => mmtMinStageTwoMaxIdOne
           );

MinMaxCompStageTwoMaxIdTwo : LdpccMinMaxComp
port map ( mmtMinMaxCompIn1    => mmtMaxStageOneIdThree,
            mmtMinMaxCompIn2    => mmtMaxStageTwoMinIdOne,
            --Outputs
            mmtMinMaxCompMinOut => mmtMinStageTwoMaxIdTwo
           );

MinMaxCompStageThreeMaxIdOne : LdpccMinMaxComp
port map ( mmtMinMaxCompIn1    => mmtMinStageTwoMaxIdOne,
            mmtMinMaxCompIn2    => mmtMaxStageTwoMinIdTwo,
            --Outputs
            mmtMinMaxCompMinOut => mmtMinStageThreeMaxIdOne
           );

MinMaxCompStageThreeMaxIdTwo : LdpccMinMaxComp
port map ( mmtMinMaxCompIn1    => mmtMinStageTwoMaxIdTwo,
            mmtMinMaxCompIn2    => mmtMinStageThreeMaxIdOne,
            --Outputs
            mmtMinMaxCompMinOut => mmtSecondMinimum_Next
           );
```

-- UPDATE_MIN_MAX_TERMS

```
 -- Update the _Next values of the delay lines based on the
 -- new inputs and the new code bit.

UPDATE_MIN_MAX_TERMS : process ( llrLlrTermIn1 , llrLlrTermIn2 , llrLlrTermIn3 ,
                                 llrLlrTermIn4 , llrLlrTermIn5 , llrLlrTermIn6 ,
                                 mmtMinMaxTermIn1 , mmtMinMaxTermIn2 , mmtMinMaxTermIn3 ,
                                 mmtMinMaxTermIn4 , mmtMinMaxTermIn5 , mmtMinMaxTermIn6 ) is
begin

     -- Augment the LLRs with the port they came in so we can
     -- do the output mapping correctly.

   mmtMinMaxTermIn1 <= (Llr=>llrLlrTermIn1 , Id=>"000");
   mmtMinMaxTermIn2 <= (Llr=>llrLlrTermIn2 , Id=>"001");
   mmtMinMaxTermIn3 <= (Llr=>llrLlrTermIn3 , Id=>"010");
   mmtMinMaxTermIn4 <= (Llr=>llrLlrTermIn4 , Id=>"011");
   mmtMinMaxTermIn5 <= (Llr=>llrLlrTermIn5 , Id=>"100");
   mmtMinMaxTermIn6 <= (Llr=>llrLlrTermIn6 , Id=>"101");

   lbSignBitTerm1_Del1_Next <= llrLlrTermIn1.Sgn;
   lbSignBitTerm2_Del1_Next <= llrLlrTermIn2.Sgn;
   lbSignBitTerm3_Del1_Next <= llrLlrTermIn3.Sgn;
   lbSignBitTerm4_Del1_Next <= llrLlrTermIn4.Sgn;
   lbSignBitTerm5_Del1_Next <= llrLlrTermIn5.Sgn;
   lbSignBitTerm6_Del1_Next <= llrLlrTermIn6.Sgn;

end process UPDATE_MIN_MAX_TERMS;
```

-- UPDATE_MAGNITUDES

```
   -- Write the minimum value into all the output registers except for
   -- the one corresponding to the position of the minima. It gets the
   -- second minima.

UPDATE_MAGNITUDES : process ( mmtMinimum , mmtSecondMinimum ) is
begin
```

```
    if ( mmtMinimum.Id = "000" ) then
       llrLlrTermOut1Reg_Next.Mag <= mmtSecondMinimum.Llr.Mag;
       llrLlrTermOut1Reg_Next.Sat <= mmtSecondMinimum.Llr.Sat;
    else
       llrLlrTermOut1Reg_Next.Mag <= mmtMinimum.Llr.Mag;
       llrLlrTermOut1Reg_Next.Sat <= mmtMinimum.Llr.Sat;
    end if;
    if ( mmtMinimum.Id = "001" ) then
       llrLlrTermOut2Reg_Next.Mag <= mmtSecondMinimum.Llr.Mag;
       llrLlrTermOut2Reg_Next.Sat <= mmtSecondMinimum.Llr.Sat;
    else
       llrLlrTermOut2Reg_Next.Mag <= mmtMinimum.Llr.Mag;
       llrLlrTermOut2Reg_Next.Sat <= mmtMinimum.Llr.Sat;
    end if;
    if ( mmtMinimum.Id = "010" ) then
       llrLlrTermOut3Reg_Next.Mag <= mmtSecondMinimum.Llr.Mag;
       llrLlrTermOut3Reg_Next.Sat <= mmtSecondMinimum.Llr.Sat;
    else
       llrLlrTermOut3Reg_Next.Mag <= mmtMinimum.Llr.Mag;
       llrLlrTermOut3Reg_Next.Sat <= mmtMinimum.Llr.Sat;
    end if;
    if ( mmtMinimum.Id = "011" ) then
       llrLlrTermOut4Reg_Next.Mag <= mmtSecondMinimum.Llr.Mag;
       llrLlrTermOut4Reg_Next.Sat <= mmtSecondMinimum.Llr.Sat;
    else
       llrLlrTermOut4Reg_Next.Mag <= mmtMinimum.Llr.Mag;
       llrLlrTermOut4Reg_Next.Sat <= mmtMinimum.Llr.Sat;
    end if;
    if ( mmtMinimum.Id = "100" ) then
       llrLlrTermOut5Reg_Next.Mag <= mmtSecondMinimum.Llr.Mag;
       llrLlrTermOut5Reg_Next.Sat <= mmtSecondMinimum.Llr.Sat;
    else
       llrLlrTermOut5Reg_Next.Mag <= mmtMinimum.Llr.Mag;
       llrLlrTermOut5Reg_Next.Sat <= mmtMinimum.Llr.Sat;
    end if;
    if ( mmtMinimum.Id = "101" ) then
       llrLlrTermOut6Reg_Next.Mag <= mmtSecondMinimum.Llr.Mag;
       llrLlrTermOut6Reg_Next.Sat <= mmtSecondMinimum.Llr.Sat;
    else
       llrLlrTermOut6Reg_Next.Mag <= mmtMinimum.Llr.Mag;
       llrLlrTermOut6Reg_Next.Sat <= mmtMinimum.Llr.Sat;
    end if;

 end process UPDATE_MAGNITUDES;

-------------------------------------------------------------------------------
-- UPDATE_SIGNS
-------------------------------------------------------------------------------

 llrLlrTermIn1_Sgn_tmp <= llrLlrTermIn1.Sgn;
 llrLlrTermIn2_Sgn_tmp <= llrLlrTermIn2.Sgn;
 llrLlrTermIn3_Sgn_tmp <= llrLlrTermIn3.Sgn;
 llrLlrTermIn4_Sgn_tmp <= llrLlrTermIn4.Sgn;
 llrLlrTermIn5_Sgn_tmp <= llrLlrTermIn5.Sgn;
 llrLlrTermIn6_Sgn_tmp <= llrLlrTermIn6.Sgn;

 UPDATE_SIGNS : process ( llrLlrTermIn1_Sgn_tmp , llrLlrTermIn2_Sgn_tmp , llrLlrTermIn3_Sgn_tmp ,
                          llrLlrTermIn4_Sgn_tmp , llrLlrTermIn5_Sgn_tmp , llrLlrTermIn6_Sgn_tmp ,
                          lbTotalSign ,
                          lbSignBitTerm1_Del1 , lbSignBitTerm2_Del1 , lbSignBitTerm3_Del1 ,
                          lbSignBitTerm4_Del1 , lbSignBitTerm5_Del1 , lbSignBitTerm6_Del1 ) is
 begin

      -- Determine the total sign as the XOR of all the input sign
      -- bits. We pipeline this signal.

    lbTotalSign_Next <= llrLlrTermIn1_Sgn_tmp xor llrLlrTermIn2_Sgn_tmp xor
                        llrLlrTermIn3_Sgn_tmp xor llrLlrTermIn4_Sgn_tmp xor
                        llrLlrTermIn5_Sgn_tmp xor llrLlrTermIn6_Sgn_tmp;

      -- The output signs are corrected by xor ing the total sign
      -- by their own input sign. Note the pipelining.

    llrLlrTermOut1Reg_Next.Sgn <= lbTotalSign xor lbSignBitTerm1_Del1;
    llrLlrTermOut2Reg_Next.Sgn <= lbTotalSign xor lbSignBitTerm2_Del1;
    llrLlrTermOut3Reg_Next.Sgn <= lbTotalSign xor lbSignBitTerm3_Del1;
    llrLlrTermOut4Reg_Next.Sgn <= lbTotalSign xor lbSignBitTerm4_Del1;
    llrLlrTermOut5Reg_Next.Sgn <= lbTotalSign xor lbSignBitTerm5_Del1;
    llrLlrTermOut6Reg_Next.Sgn <= lbTotalSign xor lbSignBitTerm6_Del1;

 end process UPDATE_SIGNS;

   -- Take a registered output to help with timing closure and
   -- to be nice to the next block.

-------------------------------------------------------------------------------
-- ASSIGN_OUTPUT
-------------------------------------------------------------------------------
```

```
  ASSIGN_OUTPUT : process ( llrLlrTermOut1Reg , llrLlrTermOut2Reg , llrLlrTermOut3Reg ,
                            llrLlrTermOut4Reg , llrLlrTermOut5Reg , llrLlrTermOut6Reg   ) is
  begin
    llrLlrTermOut1  <= llrLlrTermOut1Reg;
    llrLlrTermOut2  <= llrLlrTermOut2Reg;
    llrLlrTermOut3  <= llrLlrTermOut3Reg;
    llrLlrTermOut4  <= llrLlrTermOut4Reg;
    llrLlrTermOut5  <= llrLlrTermOut5Reg;
    llrLlrTermOut6  <= llrLlrTermOut6Reg;

  end process ASSIGN_OUTPUT;

-------------------------------------------------------------------------------
-- CLOCK_UPDATE
-------------------------------------------------------------------------------

    -- This is the process which updates all the registers
    -- with their new values.

  CLOCK_UPDATE : process ( clkClock ) is
  begin

      if ( clkClock'event and clkClock='1' ) then

       if ( rstReset = '1' ) then

          mmtSecondMinimum <= MIN_MAX_TERM_RESET;
          mmtMinimum       <= MIN_MAX_TERM_RESET;

          llrLlrTermOut1Reg   <= LOG_LIKE_RATIO_RESET;
          llrLlrTermOut2Reg   <= LOG_LIKE_RATIO_RESET;
          llrLlrTermOut3Reg   <= LOG_LIKE_RATIO_RESET;
          llrLlrTermOut4Reg   <= LOG_LIKE_RATIO_RESET;
          llrLlrTermOut5Reg   <= LOG_LIKE_RATIO_RESET;
          llrLlrTermOut6Reg   <= LOG_LIKE_RATIO_RESET;

          lbSignBitTerm1_Del1 <= '0';
          lbSignBitTerm2_Del1 <= '0';
          lbSignBitTerm3_Del1 <= '0';
          lbSignBitTerm4_Del1 <= '0';
          lbSignBitTerm5_Del1 <= '0';
          lbSignBitTerm6_Del1 <= '0';

          lbTotalSign <= '0';

        else
          mmtSecondMinimum    <= mmtSecondMinimum_Next;
          mmtMinimum          <= mmtMinimum_Next;

          llrLlrTermOut1Reg   <= llrLlrTermOut1Reg_Next;
          llrLlrTermOut2Reg   <= llrLlrTermOut2Reg_Next;
          llrLlrTermOut3Reg   <= llrLlrTermOut3Reg_Next;
          llrLlrTermOut4Reg   <= llrLlrTermOut4Reg_Next;
          llrLlrTermOut5Reg   <= llrLlrTermOut5Reg_Next;
          llrLlrTermOut6Reg   <= llrLlrTermOut6Reg_Next;

          lbSignBitTerm1_Del1 <= lbSignBitTerm1_Del1_Next;
          lbSignBitTerm2_Del1 <= lbSignBitTerm2_Del1_Next;
          lbSignBitTerm3_Del1 <= lbSignBitTerm3_Del1_Next;
          lbSignBitTerm4_Del1 <= lbSignBitTerm4_Del1_Next;
          lbSignBitTerm5_Del1 <= lbSignBitTerm5_Del1_Next;
          lbSignBitTerm6_Del1 <= lbSignBitTerm6_Del1_Next;

          lbTotalSign <= lbTotalSign_Next;

        end if;

       end if;

  end process CLOCK_UPDATE;

end Behavioral;
```

Listing B.11: LdpccMinMaxComp.vhd

```
-------------------------------------------------------------------------------
--
--    Filename: LdpccMinMaxComp.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
```

```
--  ------------------------------------------------------------------
--
--    Description
--    -----------
--
--  This is a simple combinatorial block that outputs the min and max of its
--  two inputs.
--
------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity LdpccMinMaxComp is
    Port ( mmtMinMaxCompIn1   : in MinMaxTerm;
           mmtMinMaxCompIn2   : in MinMaxTerm;
           --Outputs
           mmtMinMaxCompMinOut : out MinMaxTerm;
           mmtMinMaxCompMaxOut : out MinMaxTerm
         );
end LdpccMinMaxComp;

architecture Behavioral of LdpccMinMaxComp is

begin

     -- Note this is a combinatorial block only. First check the infinity bit
     -- of the first input and if set then write the other as minimum.

   FIND_MINIMUN : process (mmtMinMaxCompIn1,mmtMinMaxCompIn2) is
   begin

      if ( mmtMinMaxCompIn1.Llr.Sat = '1' ) then

         mmtMinMaxCompMinOut <= mmtMinMaxCompIn2;
         mmtMinMaxCompMaxOut <= mmtMinMaxCompIn1;

       else

         if ( mmtMinMaxCompIn2.Llr.Sat = '1' ) then

            mmtMinMaxCompMinOut <= mmtMinMaxCompIn1;
            mmtMinMaxCompMaxOut <= mmtMinMaxCompIn2;

         else

            if ( mmtMinMaxCompIn1.Llr.Mag > mmtMinMaxCompIn2.Llr.Mag ) then
               mmtMinMaxCompMinOut <= mmtMinMaxCompIn2;
               mmtMinMaxCompMaxOut <= mmtMinMaxCompIn1;
            else
               mmtMinMaxCompMinOut <= mmtMinMaxCompIn1;
               mmtMinMaxCompMaxOut <= mmtMinMaxCompIn2;
            end if;

         end if;

      end if;

   end process FIND_MINIMUN;

end Behavioral;
```

Listing B.12: LdpccLogLikeAdd.vhd

```
------------------------------------------------------------------
--
--    Filename: LdpccLogLikeAdd.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
--  ------------------------------------------------------------------
--
--    Description
--    -----------
--
--  This is a simple combinatorial block that performs the addition phase of the
--  LLR updates..
```

```
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity LdpccLogLikeAdd is
    Port ( clkClock                 : in LogicBit;
           rstReset                 : in LogicBit;
           llrduInfoLlrDelayUnitIn  : in LogLikeRatioDelayUnit;
           llrduInfoLlrDelayUnitOut : out LogLikeRatioDelayUnit
         );
end LdpccLogLikeAdd;

architecture Behavioral of LdpccLogLikeAdd is

-------------------------------------------------------------------------------
-- COMPONENTS
-------------------------------------------------------------------------------

    -- Components which we instantiate within the main
    -- block.

component LlrSignMagAdd is
    Port ( llrLlrIn1  : in  LogLikeRatio;
           llrLlrIn2  : in  LogLikeRatio;
           llrLlrOut1 : out LogLikeRatio
           );
end component;

-------------------------------------------------------------------------------
-- REGISTERS
-------------------------------------------------------------------------------

    -- Registered values which must have an _Next value
    -- to work well. As per Massana design rules.

signal llrSumOneTwo_Next    : LogLikeRatio;
signal llrSumOneTwo         : LogLikeRatio;

signal llrSumOneThree_Next  : LogLikeRatio;
signal llrSumOneThree       : LogLikeRatio;

signal llrSumTwoThree_Next  : LogLikeRatio;
signal llrSumTwoThree       : LogLikeRatio;

signal llrChannelLlr_Del1_Next  : LogLikeRatio;
signal llrChannelLlr_Del1       : LogLikeRatio;

signal llrduInfoLlrDelayUnitOutReg_Next  : LogLikeRatioDelayUnit;
signal llrduInfoLlrDelayUnitOutReg       : LogLikeRatioDelayUnit;

-- This signal is defined to eliminate WARNINGS during synthesis
signal llrduInfoLlrDelayUnitIn_tmp : LogLikeRatio;

begin

-------------------------------------------------------------------------------
-- INSTANTIATIONS
-------------------------------------------------------------------------------

LlrSignMagAdd12 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrduInfoLlrDelayUnitIn(1),
        llrLlrIn2  => llrduInfoLlrDelayUnitIn(2),
        llrLlrOut1 => llrSumOneTwo_Next
      );

LlrSignMagAdd13 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrduInfoLlrDelayUnitIn(1),
        llrLlrIn2  => llrduInfoLlrDelayUnitIn(3),
        llrLlrOut1 => llrSumOneThree_Next
      );

LlrSignMagAdd23 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrduInfoLlrDelayUnitIn(2),
        llrLlrIn2  => llrduInfoLlrDelayUnitIn(3),
        llrLlrOut1 => llrSumTwoThree_Next
      );

LlrSignMagAdd012 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrChannelLlr_Del1,
```

```
        llrLlrIn2  => llrSumOneTwo,
        llrLlrOut1 => llrduInfoLlrDelayUnitOutReg_Next(3)
      );

LlrSignMagAdd013 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrChannelLlr_Del1,
        llrLlrIn2  => llrSumOneThree,
        llrLlrOut1 => llrduInfoLlrDelayUnitOutReg_Next(2)
      );

LlrSignMagAdd023 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrChannelLlr_Del1,
        llrLlrIn2  => llrSumTwoThree,
        llrLlrOut1 => llrduInfoLlrDelayUnitOutReg_Next(1)
      );

-------------------------------------------------------------------------------
-- CHANNEL_LLR_UPDATE
-------------------------------------------------------------------------------
--signal llrduInfoLlrDelayUnitIn_tmp : LogLikeRatio;
llrduInfoLlrDelayUnitIn_tmp <= llrduInfoLlrDelayUnitIn(0);
   CHANNEL_LLR_UPDATE : process ( llrChannelLlr_Del1 , llrduInfoLlrDelayUnitIn_tmp ) is
   begin
      llrChannelLlr_Del1_Next             <= llrduInfoLlrDelayUnitIn_tmp;
      llrduInfoLlrDelayUnitOutReg_Next(0) <= llrChannelLlr_Del1;
   end process CHANNEL_LLR_UPDATE;

-------------------------------------------------------------------------------
-- ASSIGN_OUTPUT
-------------------------------------------------------------------------------

   ASSIGN_OUTPUT : process ( llrduInfoLlrDelayUnitOutReg ) is
   begin

      llrduInfoLlrDelayUnitOut <= llrduInfoLlrDelayUnitOutReg;

   end process ASSIGN_OUTPUT;

-------------------------------------------------------------------------------
-- CLOCK_UPDATE
-------------------------------------------------------------------------------

     -- This is the process which updates all the registers
     -- with their new values.

   CLOCK_UPDATE : process ( clkClock ) is
   begin

      if ( clkClock'event and clkClock='1' ) then

         if ( rstReset = '1' ) then

            llrSumOneTwo   <= LOG_LIKE_RATIO_RESET;
            llrSumTwoThree <= LOG_LIKE_RATIO_RESET;
            llrSumOneThree <= LOG_LIKE_RATIO_RESET;

            llrChannelLlr_Del1          <= LOG_LIKE_RATIO_RESET;
            llrduInfoLlrDelayUnitOutReg <= LOG_LIKE_RATIO_DELAY_UNIT_RESET;

         else

            llrSumOneTwo   <= llrSumOneTwo_Next;
            llrSumOneThree <= llrSumOneThree_Next;
            llrSumTwoThree <= llrSumTwoThree_Next;

            llrChannelLlr_Del1          <= llrChannelLlr_Del1_Next;
            llrduInfoLlrDelayUnitOutReg <= llrduInfoLlrDelayUnitOutReg_Next;

         end if;

      end if;

   end process CLOCK_UPDATE;

end Behavioral;
```

Listing B.13: LlrSignMagAdd.vhd

```
--
--    Filename: LlrSignMagAdd.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
```

```
--    Author:    Stephen Bates & Ramkrishna Swamy
--    Location:  Dept. of Electrical and Computer Engineering
--               The University of Alberta
--               Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------
--
--   Description
--   -----------
--
-- This is a simple combinatorial block that performs the addition phase of the
-- LLR updates.
--
-- ------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity LlrSignMagAdd is
    Port ( llrLlrIn1  : in  LogLikeRatio;
           llrLlrIn2  : in  LogLikeRatio;
           llrLlrOut1 : out LogLikeRatio
         );
end LlrSignMagAdd;

architecture Behavioral of LlrSignMagAdd is

signal slvSum : std_logic_vector ( llrLlrIn1.Mag'left(1)+1 downto 0 );
signal slvIn1 : std_logic_vector ( llrLlrIn1.Mag'left(1)+1 downto 0 );
signal slvIn2 : std_logic_vector ( llrLlrIn1.Mag'left(1)+1 downto 0 );

begin

     -- Note this is a combinatorial block only..

   ADD_TERMS : process (llrLlrIn1,llrLlrIn2,slvSum,slvIn1,slvIn2) is
   begin

       -- Add the temp terms regardless of if statements below.

      slvIn1(slvIn1'left(1))           <= '0';
      slvIn1(slvIn1'left(1)-1 downto 0) <= llrLlrIn1.Mag;
      slvIn2(slvIn2'left(1))           <= '0';
      slvIn2(slvIn2'left(1)-1 downto 0) <= llrLlrIn2.Mag;

      slvSum <= slvIn1+slvIn2;

       -- If either input is +inf then the output is also + inf.

    --  The following line does not work with 'others=>'
    --  llrLlrOut1 <= ( Sat=>'1' , Sgn=>'0' , Mag=>(others=>'0') );
    --  Therefore, zeros were hardcoded for synthesis
      if ( (llrLlrIn1.Sat = '1') or (llrLlrIn2.Sat = '1') ) then
-- TB         llrLlrOut1 <= ( Sat=>'1' , Sgn=>'0' , Mag=>(others=>'0') );
        llrLlrOut1 <= ( Sat=>'1' , Sgn=>'0' , Mag=>"0000" );
      else

        llrLlrOut1.Sat <= '0';

          -- Add or sutract the magnitudes as given
          -- by the sign bit. Note if we add we check for
          -- saturation. If we subtract we must put the larger
          -- first.

        if (llrLlrIn1.Sgn = llrLlrIn2.Sgn ) then

          llrLlrOut1.Sgn                    <= llrLlrIn1.Sgn;

          if ( slvSum(slvSum'left(1)) = '1' ) then
-- TB             llrLlrOut1.Mag <= (others=>'1');
            llrLlrOut1.Mag <= "1111";
          else
            llrLlrOut1.Mag <= slvSum(slvSum'left(1)-1 downto 0);
          end if;
        else

          if ( llrLlrIn1.Mag > llrLlrIn2.Mag ) then
            llrLlrOut1.Sgn <= llrLlrIn1.Sgn;
            llrLlrOut1.Mag <= llrLlrIn1.Mag - llrLlrIn2.Mag;
          else
            llrLlrOut1.Sgn <= llrLlrIn2.Sgn;
            llrLlrOut1.Mag <= llrLlrIn2.Mag - llrLlrIn1.Mag;
          end if;

         end if;
```

```
      end if;

   end process ADD_TERMS;

end Behavioral;
```

Listing B.14: LdpccSignExtractor.vhd

```
--------------------------------------------------------------------------------
--
--    Filename: LdpccSignExtractor.vhd
--
--    Created:  29th April 2004
--    Modified: 29th April 2004
--    Version:  0.1
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
-- ------------------------------------------------------------------------
--
--    Description
--    -----------
--
--  This is the hard-slicer unit in the LDPC-CC decoder.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;


entity LdpccSignExtractor is
  Port ( clkClock              : in LogicBit;
         rstReset              : in LogicBit;
         llrduLlrDelayUnitIn   : in LogLikeRatioDelayUnit;
         --Outputs
         lbBitOut              : out LogicBit
         );
end LdpccSignExtractor;

architecture Behavioral of LdpccSignExtractor is

--------------------------------------------------------------------------------
-- COMPONENTS
--------------------------------------------------------------------------------

    -- Components which we instantiate within the main
    -- block.

component LlrSignMagAdd is
    Port ( llrLlrIn1  : in  LogLikeRatio;
           llrLlrIn2  : in  LogLikeRatio;
           llrLlrOut1 : out LogLikeRatio
              );
end component;


--------------------------------------------------------------------------------
-- REGISTERS
--------------------------------------------------------------------------------

    -- Registered values which must have an _Next value
    -- to work well. As per Massana design rules.

signal llrSumZeroOne_Next    : LogLikeRatio;
signal llrSumZeroOne         : LogLikeRatio;

signal llrSumZeroOneTwoThree_Next : LogLikeRatio;
signal llrSumZeroOneTwoThree      : LogLikeRatio;

signal llrSumTwoThree_Next : LogLikeRatio;
signal llrSumTwoThree      : LogLikeRatio;


begin

--------------------------------------------------------------------------------
-- INSTANTIATIONS
--------------------------------------------------------------------------------
```

```
  LlrSignMagAdd01 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrduLlrDelayUnitIn(0),
        llrLlrIn2  => llrduLlrDelayUnitIn(1),
        llrLlrOut1 => llrSumZeroOne_Next
      );

  LlrSignMagAdd23 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrduLlrDelayUnitIn(2),
        llrLlrIn2  => llrduLlrDelayUnitIn(3),
        llrLlrOut1 => llrSumTwoThree_Next
      );

  LlrSignMagAdd0123 : LlrSignMagAdd
   port map
      ( llrLlrIn1  => llrSumZeroOne,
        llrLlrIn2  => llrSumTwoThree,
        llrLlrOut1 => llrSumZeroOneTwoThree_Next
      );

-------------------------------------------------------------------------
-- ASSIGN_OUTPUT
-------------------------------------------------------------------------

   ASSIGN_OUTPUT : process ( llrSumZeroOneTwoThree ) is
   begin

      lbBitOut <= llrSumZeroOneTwoThree.Sgn;

   end process ASSIGN_OUTPUT;


-------------------------------------------------------------------------
-- CLOCK_UPDATE
-------------------------------------------------------------------------

     -- This is the process which updates all the registers
     -- with their new values.

   CLOCK_UPDATE : process ( clkClock ) is
   begin

      if ( clkClock'event and clkClock='1' ) then

         if ( rstReset = '1' ) then

            llrSumZeroOne          <= LOG_LIKE_RATIO_RESET;
            llrSumTwoThree         <= LOG_LIKE_RATIO_RESET;
            llrSumZeroOneTwoThree <= LOG_LIKE_RATIO_RESET;
         else
            llrSumZeroOne          <= llrSumZeroOne_Next;
            llrSumZeroOneTwoThree <= llrSumZeroOneTwoThree_Next;
            llrSumTwoThree         <= llrSumTwoThree_Next;

     end if;

      end if;

   end process CLOCK_UPDATE;

end Behavioral;
```

Listing B.15: ErrorCounter.vhd

```
----------------------------------------------------------------------
--
--    Filename: ErrorCounter.vhd
--
--    Created:  06 July 2004
--    Modified: 01 September 2004
--    Version:  $
--
--    Author:   Stephen Bates & Ramkrishna Swamy
--    Location: Dept. of Electrical and Computer Engineering
--              The University of Alberta
--              Edmonton, Alberta, T6G 2V4.
--
--  ------------------------------------------------------------------
--
--   Description
--   -----------
--
--  This function tracks the BER and outputs an 8 segment display signal to
--  report this.
----------------------------------------------------------------------

library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.LdpccPackageForIcfaalp2.ALL;

entity ErrorCounter is
  port (
      -- inputs
      clkClock             : in  std_logic;          -- clock
      rstReset             : in  std_logic;          -- active high reset
      lbDecodedInfoBit     : in  LogicBit;
      -- outputs
      slvBlockCountLog2    : out std_logic_vector (7 downto 0));
end ErrorCounter;

architecture Behavourial of ErrorCounter is

signal bcBlockCounter               : BlockCounter;
signal bcBlockCounter_Next          : BlockCounter;
signal bcThisBlockCounterMax        : BlockCounter;
signal bcThisBlockCounterMax_Next   : BlockCounter;

signal bcErrorCount: BlockCounter;
signal bcErrorCount_Next: BlockCounter;
signal bcErrorCount01: BlockCounter;
signal bcErrorCount01_Next: BlockCounter;
signal bcErrorCount10: BlockCounter;
signal bcErrorCount10_Next: BlockCounter;
signal lbBitToCompare10: LogicBit;
signal lbBitToCompare10_Next: LogicBit;
begin

-------------------------------------------------------------------------
-- ASSIGN_OUTPUT
-------------------------------------------------------------------------

    ASSIGN_OUTPUT : process( bcThisBlockCounterMax ) is
    begin

  -- We can do this as a slv or as a driver for a 7 segment
  -- display.

    case bcThisBlockCounterMax is
    when ERROR_WIN_MIN =>
      slvBlockCountLog2 <= "11111001";--'1'
    when "00000000000000000000000000001000000000000" =>
      slvBlockCountLog2 <= "10100100";--'2'
    when "00000000000000000000000000010000000000000" =>
      slvBlockCountLog2 <= "10110000";--'3'
    when "00000000000000000000000000100000000000000" =>
      slvBlockCountLog2 <= "10011001";--'4'
    when "00000000000000000000000001000000000000000" =>
      slvBlockCountLog2 <= "10010010";--'5'
    when "00000000000000000000000010000000000000000" =>
      slvBlockCountLog2 <= "10000010";--'6'
    when "00000000000000000000000100000000000000000" =>
      slvBlockCountLog2 <= "11111000";--'7'
    when "00000000000000000000001000000000000000000" =>
      slvBlockCountLog2 <= "10000000";--'8'
    when "00000000000000000000010000000000000000000" =>
      slvBlockCountLog2 <= "10010000";--'9'
    when "00000000000000000000100000000000000000000" =>
      slvBlockCountLog2 <= "10001000";--'A'
    when "00000000000000000001000000000000000000000" =>
      slvBlockCountLog2 <= "10000011";--'B'
    when "00000000000000000010000000000000000000000" =>
      slvBlockCountLog2 <= "11000110";--'C'
    when "00000000000000000100000000000000000000000" =>
      slvBlockCountLog2 <= "10100001";--'D'
    when "00000000000000001000000000000000000000000" =>
      slvBlockCountLog2 <= "10000110";--'E'
    when "00000000000000010000000000000000000000000" =>
      slvBlockCountLog2 <= "10001110";--'F'
    when "00000000000000100000000000000000000000000" =>
      slvBlockCountLog2 <= "01000000";--'0.'
    when "00000000000001000000000000000000000000000" =>
      slvBlockCountLog2 <= "01111001";--'1.'
    when "00000000000010000000000000000000000000000" =>
      slvBlockCountLog2 <= "00100100";--'2.'
    when "00000000000100000000000000000000000000000" =>
      slvBlockCountLog2 <= "00110000";--'3.'
    when "00000000001000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00011001";--'4.'
    when "00000000010000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00010010";--'5.'
    when "00000000100000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00000010";--'6.'
    when "00000001000000000000000000000000000000000" =>
```

```
      slvBlockCountLog2 <= "01111000";-- '7.'
    when "00000001000000000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00000000";-- '8.'
    when "00000010000000000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00010000";-- '9.'
    when "00000100000000000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00001000";-- 'A.'
    when "00001000000000000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00000011";-- 'B.'
    when "00010000000000000000000000000000000000000000" =>
      slvBlockCountLog2 <= "01000110";-- 'C.'
    when "00100000000000000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00100001";-- 'D.'
    when "01000000000000000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00000110";-- 'E.'
    when "10000000000000000000000000000000000000000000" =>
      slvBlockCountLog2 <= "00001110";-- 'F.'
    when others =>
      slvBlockCountLog2 <= "01001111";-- 'Err'
    end case;

    end process ASSIGN_OUTPUT;

-------------------------------------------------------------------------------
-- BLOCK_COUNTER_MODE
-------------------------------------------------------------------------------

  -- In this mode of operation we find the counter period 2^N that is
  -- closest to giving us NUM_ERRORS_WANTED errors in its period. It
  -- is adaptive in that the period increments or decrements until
  -- it gets close to the desired point (or saturates). Note the period
  -- can lie in the range ERROR_WIN_MIN to ERROR_WIN_MAX.

    BLOCK_COUNTER_MODE : process (bcThisBlockCounterMax, bcErrorCount, bcBlockCounter)
    begin

    if ( bcBlockCounter = bcThisBlockCounterMax ) then


       -- Set the counter upper value based on the number of
       -- errors observed. Saturate at our limits.

      if ( bcErrorCount > NUM_ERRORS_WANTED_UPPER_THRESH ) then

        if ( bcThisBlockCounterMax = ERROR_WIN_MIN ) then
          bcThisBlockCounterMax_Next <= bcThisBlockCounterMax;
        else
          bcThisBlockCounterMax_Next(bcThisBlockCounterMax'left(1))              <= '0';
          bcThisBlockCounterMax_Next(bcThisBlockCounterMax'left(1)-1 downto 0) <=
                bcThisBlockCounterMax(bcThisBlockCounterMax'left(1) downto 1);
        end if;


      elsif ( bcErrorCount < NUM_ERRORS_WANTED_LOWER_THRESH ) then

        if ( bcThisBlockCounterMax = ERROR_WIN_MAX ) then
          bcThisBlockCounterMax_Next <= bcThisBlockCounterMax;
        else
          bcThisBlockCounterMax_Next(bcThisBlockCounterMax'left(1) downto 1) <=
                bcThisBlockCounterMax(bcThisBlockCounterMax'left(1)-1 downto 0);
          bcThisBlockCounterMax_Next(0)                                          <= '0';
        end if;

      else

        bcThisBlockCounterMax_Next <= bcThisBlockCounterMax;

      end if;

        -- Latch the error count result to the output. That
        -- will be held untile next eval of this loop.

      bcBlockCounter_Next       <= (others=>'0');

    else
      bcBlockCounter_Next         <= bcBlockCounter + BLOCK_COUNTER_INC;
      bcThisBlockCounterMax_Next <= bcThisBlockCounterMax;
    end if;

    end process BLOCK_COUNTER_MODE;

-------------------------------------------------------------------------------
-- ERROR_TRACKING
-------------------------------------------------------------------------------

  -- Generate an alterating sequence and compare for both possible phases wrt
  -- that. One will be right and one will be wrong. The compare could be an issue
  -- so there are clever ways to do it.
```

```
    ERROR_TRACKING : process (lbBitToCompare10, bcBlockCounter, bcThisBlockCounterMax,
                              bcErrorCount01, bcErrorCount10, lbDecodedInfoBit) is
    begin

      lbBitToCompare10_Next <= not lbBitToCompare10;

     if ( bcBlockCounter = bcThisBlockCounterMax ) then

      bcErrorCount01_Next <= (others=>'0');
      bcErrorCount10_Next <= (others=>'0');
      bcErrorCount_Next   <= (others=>'0');

     else

      if ( lbDecodedInfoBit = lbBitToCompare10 ) then
        bcErrorCount01_Next <= bcErrorCount01 + BLOCK_COUNTER_INC;
        bcErrorCount10_Next <= bcErrorCount10;
      else
        bcErrorCount01_Next <= bcErrorCount01;
        bcErrorCount10_Next <= bcErrorCount10 + BLOCK_COUNTER_INC;
      end if;

      if ( bcErrorCount01 < bcErrorCount10 ) then
        bcErrorCount_Next <= bcErrorCount01;
      else
        bcErrorCount_Next <= bcErrorCount10;
      end if;

     end if;

   end process ERROR_TRACKING;

-------------------------------------------------------------------------------
-- CLOCK_UPDATE
-------------------------------------------------------------------------------

    CLOCK_UPDATE : process ( clkClock ) is
    begin

     if ( clkClock'event and clkClock='1' ) then

            if ( rstReset = '1' ) then

              bcBlockCounter         <= (others=>'0');
              bcThisBlockCounterMax  <= ERROR_WIN_MIN;

              bcErrorCount           <= (others=>'0');
              bcErrorCount01         <= (others=>'0');
              bcErrorCount10         <= (others=>'0');
              lbBitToCompare10       <= '1';
            else
              bcBlockCounter         <= bcBlockCounter_Next;
              bcThisBlockCounterMax  <= bcThisBlockCounterMax_Next;

              bcErrorCount           <=bcErrorCount_Next;
              bcErrorCount01         <= bcErrorCount01_Next;
              bcErrorCount10         <= bcErrorCount10_Next;
              lbBitToCompare10       <= lbBitToCompare10_Next;

            end if;

          end if;

    end process CLOCK_UPDATE;

end Behavourial;
```

Appendix C

Tutorial: CFCL for LDPC-CCs

A new technology for pipelined VLSI, based on complementary ferroelectric capacitor logic (CFCL), was recently reported by Hanyu, Kimura, Matsunaga and others. CFCL circuits use of a layer of ferroelectric capacitor (FeC) material for non-volatile storage of data. The FeC material is deposited across the top of existing CMOS logic gates, creating an extra memory layer with negligible footprint on the CMOS layout.

FeCs can replace RAM cells and latches in CMOS designs, leading to a considerable reduction in transistor count and circuit area. Because FeCs are non-volatile, no static current is needed for data retention. CFCL circuits therefore offer extremely low static power consumption.

Kimura, Hanyu, *et al.* recently demonstrated a CFCL-based low-power content-addressable memory (CAM) circuit with **7700 times lower static power** than an equivalent CMOS design. The CFCL design also achieved a **33% reduction in dynamic power** consumption and a **66% reduction in circuit area**. This design used a 0.6 μm technology. At present, a 0.35 μm CFCL technology is available with FeCs sized 1 μm^2. CFCL logic uses differential storage, requiring two to four FeCs per gate, making FeCs quite competitive in area against static RAMs (or SRAMs) and other memory circuits.

C.1 Content-Addressable-Memory

The CFCL-based CAM designed by Kimura *et al.* is an array of bits arranged in rows and columns. The CAM performs, for each row i, a *greater-than* operation, $G(X,B_i)$ between an input word X and a set of stored words B_i, defined as

$$G(X,B_i) = \begin{cases} 1, \text{if } X > B_i \\ 0, \text{otherwise.} \end{cases} \tag{C.1}$$

This function is decomposed into pair-wise operations performed between bits in a row. Each row produces a single bit of output, z_i, as illustrated in Fig. C.1.

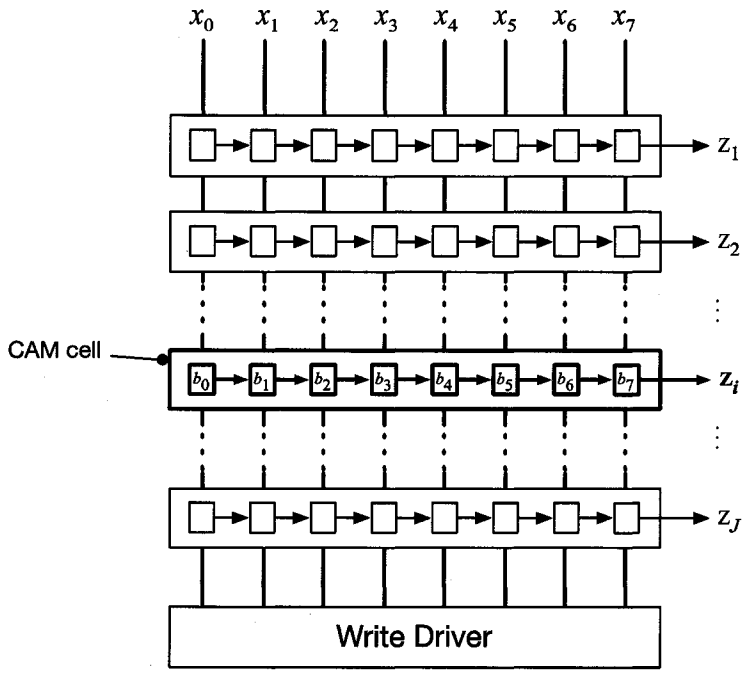


Figure C.1: Structure of the CFCL-based CAM. *[Figure courtesy of Dr. Chris Winstead, Utah State University.]*

C.2 CAM-based LDPC-CC Decoder

In this section, we present a possible CAM-based architecture for implementing LDPC-CC decoders. This architecture is not necessarily the most efficient, but it illustrates a plausible method for applying CFCL to LDPC-CC decoding.

The CAM operation (C.1) described in Section C.1 can be used to implement the vertical (parity-check) update (3.13) used in LDPC decoders. This implementation is based on sorting the stored metrics, $l_{i,j}$, for a given phase φ. The two smallest metrics are then selected as the solution to (3.13).

One advantage of this method is that the sorted order can be forwarded from one processor to the next. In most cases, the sorted order will not change after the first two iterations and the solution to (3.13) can subsequently be obtained in a single operation. The disadvantage is that additional memory is needed to store the sorted order. Since only a small fraction of CAM cells are active at any time, the static power holds a significant share of the total power used in this architecture.

C.2.1 Vertical Processing: Parity-Check Update

The CAM-based LDPC-CC decoder consists of at least m_s+1 CAM columns. Each column has $J+1$ rows. The CAM design differs from that described in Section C.1 in two ways:

- The stored words B_i can be retrieved by a read operation, i.e., the CAM has read, write and execute modes.
- Only one row is active per column at any time (in typical CAMs, all rows execute their function simultaneously on the same input word).

For one phase of vertical processing, one cell is selected in each of K columns, as indicated in Fig. C.2. Initially, one cell is read, and the retrieved metric is called the *test metric*, Λ. The remaining $K-1$ cells are the *sort space*, S. The test metric is offered as input to all cells in the sort space. Each cell q in S is executed on input Λ, producing the comparison bit z_q and the sign bit s_q. XOR operations are used on the s_q to determine the updated sign bits.

An updated sort space, S', is then created by excluding from S all cells for which $z_q = 1$. A new test metric Λ' is selected from S', and the sort operation is repeated. This iterative sort algorithm is illustrated in Fig. C.3. When a sort space is obtained for which $|S| = 1$, we know that S contains the index q of the smallest metric, and Λ is the second smallest. The sort is then terminated. All cells in φ except q are overwritten with the metric stored at q. The cell q is overwritten with the metric Λ.

In the worst case, read, execute and write operations cannot happen during the same clock cycle, and a maximum of $K-1$ sort operations are needed. This leads to a worst-case scenario of $2K$ clocks to complete the operation. A typical $(3,6)$ code would thus require 12 clock cycles per phase.

The time lost in vertical processing can be recovered by processing several phases in parallel. Because the parity-check matrix is sparse, several adjacent phases can be processed without accessing the same columns. If the CAM operates at 64ns per operation, as reported by Kimura *et al.*, and six phases are computed in parallel, then 1-10 Mbps throughput is easily achieved.

In typical operation, the sort order does not change after the first processor for most phases. Processing activity therefore decreases rapidly after the first processor. Because dynamic processing becomes rare, low static power consumption is a critical feature in this architecture. The Bates LDPC-CC decoding architecture has 30 processors and achieves 10 nJ/bit. Much of this energy is consumed by static power in memory cells. The *7700x* reduction in static power will therefore play a critical role in improving the energy efficiency of a CAM-based LDPC-CC decoder.

C.3 More Information

The CFCL tutorial presented in this Appendix, all text and figures inclusive, is fully attributed to Dr. Chris Winstead of Utah State University. For more information, see [72].

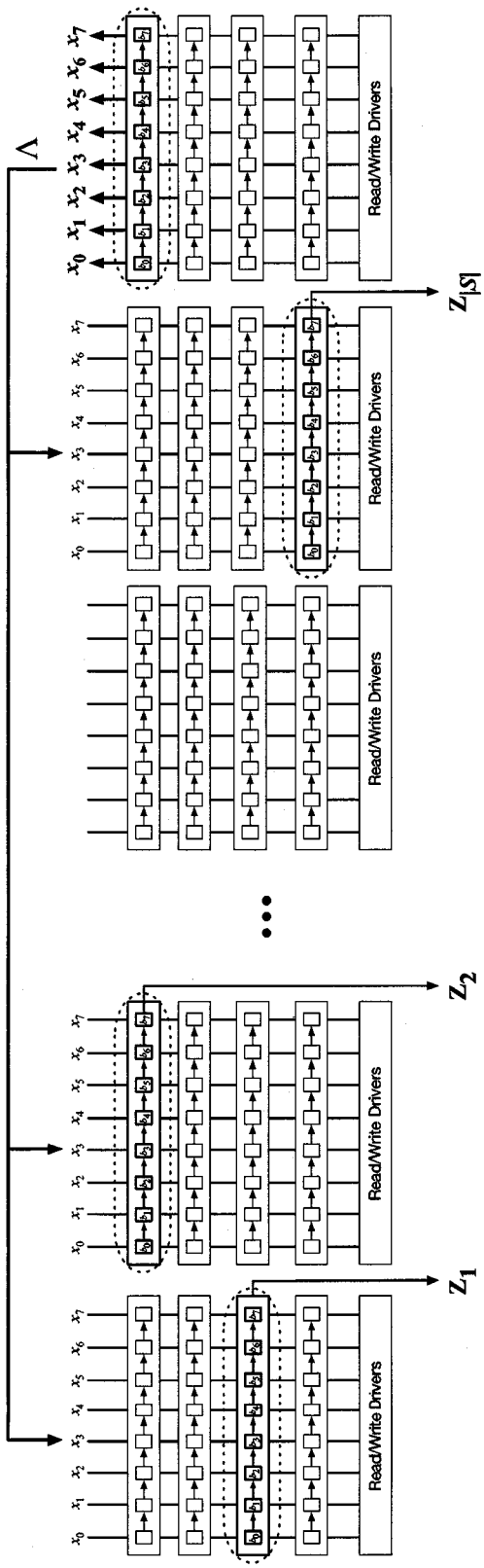


Figure C.2: CAM architecture adapted for LDPC-CC decoding. *[Figure courtesy of Dr. Chris Winstead, Utah State University.]*

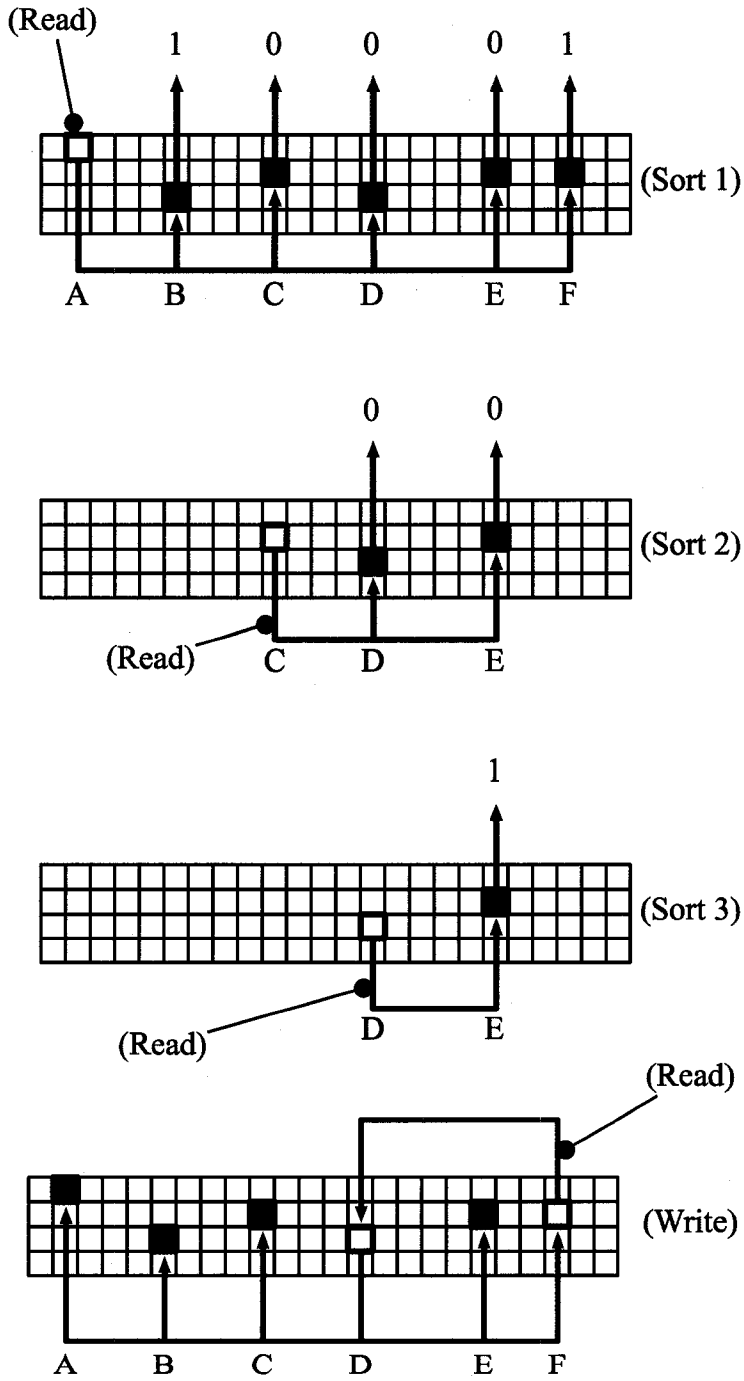


Figure C.3: Example of vertical processing in a CAM-based LDPC-CC decoder. All read, write and execute operations are indicated. *[Figure courtesy of Dr. Chris Winstead, Utah State University.]*

Appendix D

Technical Writing (2004 - 2007)

D.1 Journal Publications (accepted or in print)

- S. Bates, D. Elliott and **R. Swamy**, *"Termination Sequence Generation Circuits for Low-Density Parity-Check Convolutional Codes,"* in Proc. of the IEEE Transactions on Circuits and Systems I (TCAS I), 2006.
- **R. Swamy**, S. Bates, T. Brandon, B. Cockburn, D. Elliott, J. Koob and Z. Chen, *"Design and Test of a 175-Mbps, Rate-1/2 (128,3,6) Low-Density Parity-Check Convolutional Encoder and Decoder,"* accepted to appear in the IEEE Journal of Solid-State Circuits, 2007.

D.2 Refereed Conference Proceedings (accepted or in print)

- **R. Swamy**, S. Bates and T. Brandon, *"Architectures for ASIC implementations of low-density parity-check encoders and decoders,"* in Proc. of the IEEE Symposium on Circuits and Systems (ISCAS), Kobe, Japan, 2005.
- **R. Swamy**, M. Khorasani, Y. Liu, D. Elliott and S. Bates, *"A Fast, Pipelined Implementation of a Two-Dimensional Inverse Discrete Cosine Transform,"* in Proc. of the 18^{th} IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Saskatoon, Saskatchewan, Canada, 2005.
- **R. Swamy** and S. Bates, *"A Soft-Decision Low-Density Parity-Check Convolutional Encoder and Decoder System,"* in Proc. of the Semiconductor Research Corporation's (SRC) TECHCON 2005, Portland, Oregon, U.S.A., 2005.

- S. Bates and **R. Swamy**, *"Parallel Encoders for Low-Density Parity-Check Convolutional Codes,"* in Proc. of the IEEE Symposium on Circuits and Systems (ISCAS), Island of Kos, Greece, 2006.
- Z. Chen, **R. Swamy** and S. Bates, *"A New Encoder Implementation for Low-Density Parity-Check Convolutional Codes,"* accepted to appear in Proc. of the IEEE MWSCAS/NEWCAS Conference, Montreal, Canada, 2007.

D.3 Project Reports

- **R. Swamy**, *"The Hot Carrier Effect in today's NMOS Transistors,"* for a graduate-level Computer Engineering project course in Integrated Circuit Design, University of Alberta, September 2004.
- **R. Swamy**, M. Khorasani, Y. Liu and R. Lau, *"JPEG Decoder,"* for a graduate-level Computer Engineering project course in FPGA Design, University of Alberta, September 2004.
- S. Kasnavi, **R. Swamy**, M. Yiu, M. Phan, R. Singh, O. Shorka, P. Berube, J.N. Amaral, V.C. Gaudet, K. Iniewski, *"Implementation of a Pipelined Cache for IP Routers,"*, for a graduate-level Computer Engineering project course in Memory Design, April 2005.
- **R. Swamy**, P. Marshall and V. Gaudet, *"Characterization and High-level Design of a Differential Mode LDPC Block Code Decoder,"* for a graduate-level Computer Engineering project course in Analog Circuit Design, University of Alberta, September 2006.
- **R. Swamy**, *"Low-Density Parity-Check Convolutional Code Decoder on a J2210 Atsana SIMD Processor,"* for a graduate-level Computer Engineering project course in Vector Array Processing, University of Alberta, April 2007.