



Title:

***Analyzing and Implementing a Reed-Solomon Decoder
for Forward Error Correction in ADSL***

Project Period:

10th February 2007 – 5th June 2007

Group:

ASPI, group 1040

Group Members:

Aleksandras Šarmentovas
Paulius Ruzgys

Supervisors:

Rasmus Abildgren
Yannick Le Moullec

Number of reports printed: 5

Number of pages in report: 99

Number of pages in appendix: 20

Total number of pages: 133

Abstract:

This report presents a rapid design strategy for an efficient implementation of a Reed-Solomon (RS) decoder specified in ADSL standard ITU G.992.1 onto the Xilinx Virtex II FPGA and TigerSHARC ADSP-TS201 DSP.

ADSL is a home user-oriented modem technology that uses existing twisted-pair copper telephone lines to transport high-bandwidth data, such as multimedia and video.

The project goes through the given system (i.e., RS decoder) analysis, its modeling, simulation, selection of a particular RS decoder over another for its further analysis and implementation onto the available types of architectures.

Before the actual implementation step, it is necessary to determine which type of architecture (DSP or FPGA) is the most suitable for the execution of the selected RS decoder. For that, algorithm characterization is performed. The main idea of characterization is to extract relevant information from the given algorithm to guide the designer towards an efficient algorithm-architecture matching. To this effect, different performance metrics are efficiently used in the project to rapidly stress the proper architecture style for the given RS decoding algorithms.

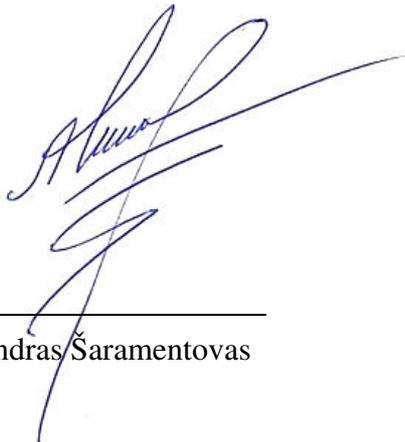
Preface

This master thesis was written by 1040 group of ASPI specialization at the Faculty of Engineering and Science, Institute of Electronic Systems, Department of Communication, Aalborg University, Denmark.

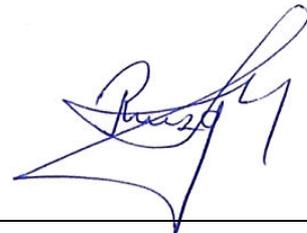
The project was proposed by the project supervisors. The project report guides the reader through a rapid design strategy for an efficient implementation of a Reed-Solomon decoder specified in ADSL standard ITU G.992.1 onto the available architectures.

The enclosed CD contains all source codes, which were used in this project work.

5th of June, 2007



Aleksandras Šarmentovas



Paulius Ruzgys

List of Abbreviations

A/D	Analog-to-Digital
ADSL	Asymmetric Digital Subscriber Line
AFI	Automatic Function Inlining
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BM	Berlekamp-Massey
BMR	Bit Manipulation Rate
BPSK	Binary Phase-Shift Keying
BTB	Branch Target Buffer
CD	Compact Disk
CDFG	Control and Data Flow Graph
CDMA	Code Division Multiple Access
CLB	Configurable Logic Block
CLU	Communications Logic Unit
CO	Central Office
CPLD	Complex Programmable Logic Device
CRC	Cyclic Redundancy Check
D/A	Digital-to-Analog
DC	Direct Current
DCM	Digital Clock Manager
DFG	Data Flow Graph
DMA	Direct Memory Access
DMT	Discrete Multi-Tone
DR	Data Ratio
DRAM	Dynamic Random Access Memory
DSL	Digital Subscriber Line
DSP	Digital Signal Processor
DVB	Digital Video Broadcasting
DVD	Digital Versatile Disc
EDAC	Error Detection and Correction
EDIF	Electronic Data Interchange Format
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
FDM	Frequency-Division Multiplexing
FEC	Forward Error Correction
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GCD	Greatest Common Divisor
GF	Galois Fields
GPP	General Purpose Processor
GRM	General Routing Matrix
HCDFG	Hierarchical Control and Data Flow Graph

HDL	Hardware Description Language
I/O	Inputs/Outputs
IAB	Instruction Alignment Buffer
IALU	Integer Arithmetic Logic Unit
IDDE	Integrated Development and Debugging Environment
IFFT	Inverse Fast Fourier Transform
IOB	Input/Output Block
IPO	Interprocedural Optimizations
ISDN	Integrated Services Digital Network
ISE	Integrated Software Environment
ITU	International Telecommunication Union
LC	Logic Cell
LFSR	Linear Feedback Shift Register
LUT	Look-Up Table
LVDS	Low-Voltage Differential Signaling
MAC	Multiply and Accumulate
MSB	The Most Significant Bit
NSP	Network Service Provider
OTP	One-Time Programmable
PC	Program Counter
PGO	Profile-Guided Optimizations
PI	Programmable Interconnection
PLD	Programmable Logic Devices
PO	Procedural Optimizations
POTS	Plain Old Telephone System
QAM	Quadrature Amplitude Modulation
RAM	Random Access Memory
RS	Reed-Solomon
SCD	Strong Circularity Degree
SHARC	Super Harvard Architecture Single-Chip Computer
SHD	Strong Harvard Degree
SMD	Strong MAC Degree
SNR	Signal-to-Noise Ratio
SOC	System-On-a-Chip
SPD	Strong Parallelism Degree
SRAM	Static Random Access Memory
TC	Transmission Convergence
TCM	Trellis Coded Modulation
VDSL	Very High Speed Digital Subscriber Line
VHDL	Very-High-Speed Integrated Circuit Hardware Description Language
WCD	Weak Circularity Degree
WHD	Weak Harvard Degree
WMD	Weak MAC Degree
WPD	Weak Parallelism Degree

Contents

1. Introduction.....	11
1.1. <i>Project Objective</i>	11
1.2. <i>A³ Framework</i>	12
1.3. <i>Design Trajectory</i>	13
1.4. <i>Project Limitations and System Constraints</i>	14
1.5. <i>System Requirements</i>	15
1.6. <i>Organization of the Report</i>	15
2. ADSL System.....	17
2.1. <i>Forward Error Correction in Transceivers</i>	17
2.1.1. <i>Error-Control Codes</i>	19
2.2. <i>Overview of ADSL System</i>	20
2.2.1. <i>Spectrum Allocation</i>	20
2.2.2. <i>ADSL Modem</i>	22
2.2.3. <i>Data Protection and Correction</i>	22
2.2.3.1. <i>Cyclic Redundancy Check (CRC)</i>	23
2.2.3.2. <i>Forward Error Correction (FEC)</i>	23
2.2.3.3. <i>Interleaving</i>	23
2.2.3.4. <i>Fast and Interleaved Paths</i>	23
2.2.4. <i>Modulation</i>	24
2.2.4.1. <i>Trellis Coded Modulation (TCM)</i>	25
2.2.4.2. <i>Bit-loading</i>	25
2.3. <i>Summary</i>	26
3. Reed-Solomon Codes	27
3.1. <i>Introduction to Reed-Solomon Codes</i>	27
3.2. <i>Properties of Reed-Solomon Codes</i>	27
3.2.1. <i>Reed-Solomon Codes Perform Well Against Burst Noise</i>	28
3.3. <i>Galois Fields</i>	29
3.3.1. <i>Properties of Finite Field</i>	29
3.3.2. <i>Prime Size Finite Field GF(p)</i>	30
3.3.2.1. <i>Binary Field GF(2)</i>	30
3.3.3. <i>Extensions to the Binary Field – GF(2^m)</i>	31
3.3.4. <i>Representation of Finite Field Elements</i>	32
3.3.5. <i>GF(2^m) Arithmetic Implementation</i>	33
3.4. <i>Reed-Solomon Encoding</i>	34
3.4.1. <i>Systematic Encoding</i>	35
3.4.2. <i>Implementation of Encoding</i>	36
3.5. <i>Reed-Solomon Decoding</i>	37
3.5.1. <i>Syndrome Calculation</i>	38
3.5.1.1. <i>Error Locations and Error Values</i>	39
3.5.2. <i>Berlekamp-Massey Algorithm</i>	40
3.5.3. <i>Euclidean Algorithm</i>	41
3.5.4. <i>Chien Search</i>	43

3.5.5. Forney Algorithm.....	44
3.6. <i>Summary</i>	44
4. Performance Evaluation of Reed-Solomon Codes.....	45
4.1. <i>Theoretical Performance of Reed-Solomon Codes</i>	45
4.1.1. Code Rate.....	45
4.1.2. Reed-Solomon Performance as a Function of Code Size	46
4.1.2.1. <i>Selection of Reed-Solomon Codeword Size</i>	47
4.1.3. Reed-Solomon Performance as a Function of Redundancy	47
4.1.4. Mis-decoding	48
4.2. <i>Simulation of Reed-Solomon Codes</i>	49
4.2.1. FEC Model.....	49
4.2.2. FEC Model Simulation	51
4.2.2.1. <i>Simulation Results</i>	51
4.2.3. Selection of Reed-Solomon Redundancy	52
4.2.3.1. <i>Advantages and Drawbacks of RS(255, 239) in ADSL</i>	52
4.3. <i>Summary</i>	53
5. Algorithm Characterization.....	55
5.1. <i>Architectural Features</i>	55
5.1.1. DSP Architectural Features.....	55
5.1.2. FPGA Architectural Features.....	56
5.2. <i>Performance Metrics</i>	56
5.2.1. Data Oriented Metric	56
5.2.2. DSP Oriented Metrics	57
5.2.2.1. <i>Circular Addressing</i>	57
5.2.2.2. <i>MAC Operations</i>	57
5.2.2.3. <i>Harvard Architecture</i>	58
5.2.3. FPGA Oriented Metrics	58
5.2.4. Difference between Strong and Weak Degrees	59
5.2.5. Selection of Defined Metrics	60
5.2.5.1. <i>Selection of FPGA Oriented Metrics</i>	60
5.2.5.2. <i>Selection of DSP Oriented Metrics</i>	65
5.2.5.3. <i>Threshold for Highly Computational Loops</i>	65
5.2.6. The Affinity	67
5.3. <i>The Design-Trotter Tool</i>	68
5.3.1. C to HCDFG Conversion.....	68
5.3.2. Algorithm Characterization in Design-Trotter.....	69
5.3.2.1. γ Metric.....	70
5.3.3. Parallelism Exploration.....	71
5.3.4. Scheduling.....	73
5.3.4.1. <i>Schedule Details</i>	73
5.3.5. Architecture Specification	73
5.4. <i>The Affinity Results</i>	74
5.5. <i>Summary</i>	75

6. FPGA Implementation	77
6.1. <i>FPGA Design Flow</i>	77
6.1.1. The ISE™ Design Flow.....	78
6.2. <i>Hardware Description Languages</i>	79
6.2.1. Handel-C	80
6.2.1.1. <i>Comparison of Handel-C and ANSI-C</i>	81
6.2.1.2. <i>Handel-C Code Optimization</i>	83
6.3. <i>Implementation Results and Analysis</i>	85
6.4. <i>Summary</i>	87
7. DSP Implementation.....	89
7.1. <i>VisualDSP++ Environment</i>	89
7.1.1. Optimizing Performance with VisualDSP++	90
7.1.2. Using the Compiler Optimizer	91
7.1.3. Tuning the Code for the Target Compiler.....	92
7.1.3.1. <i>Quad-Word-Aligning</i>	92
7.1.3.2. <i>Putting Arrays into Different Memory Blocks</i>	93
7.2. <i>Implementation Results and Analysis</i>	93
7.3. <i>DSP Implementation Results vs. FPGA Results</i>	95
7.4. <i>Summary</i>	95
8. Affinity Results Evaluation	97
8.1. <i>Evaluation of Affinity towards FPGA</i>	97
8.2. <i>Evaluation of Affinity towards DSP</i>	98
8.3. <i>General Affinity Evaluation</i>	99
8.4. <i>Cost Function</i>	101
8.5. <i>Summary</i>	103
9. Conclusion	105
9.1. <i>General Summary</i>	105
9.2. <i>Applying the Proposed Design Trajectory to Other Types of Applications...</i>	108
A. Programmable Logic	111
A.1. <i>Programmable Logic Devices (PLDs)</i>	111
A.2. <i>Basic FPGA Concepts</i>	113
A.2.1. Programming Methods.....	113
A.2.2. Look-Up Tables	114
A.2.3. FPGA Logic Block.....	115
A.3. <i>Xilinx™ Specifics</i>	116
A.3.1. Configurable Logic Blocks	117
A.3.1.1. <i>Distributed RAMs and Shift Registers</i>	119
A.3.2. RAM Blocks	119
A.3.3. Dedicated Multipliers.....	119
A.3.4. Input/Output Blocks	120
A.3.5. Digital Clock Manager.....	120
A.3.6. Programmable Routing	121

B. Architecture of TigerSHARC ADSP-TS201 Processor	123
<i>B.1. Computational Blocks</i>	<i>124</i>
B.1.1. Arithmetic Logic Unit (ALU)	124
B.1.2. Multiplier.....	125
B.1.3. Shifter	126
B.1.4. Communications Logic Unit (CLU)	127
<i>B.2. Integer ALU</i>	<i>127</i>
<i>B.3. Program Sequencer</i>	<i>127</i>
B.3.1. Instruction Line Structure.....	128
B.3.2. Instruction Alignment Buffer (IAB)	128
B.3.3. Branch Target Buffer (BTB).....	129
<i>B.4. Memory and Buses</i>	<i>129</i>
B.4.1. Buses	130
B.4.2. Memory	130
Bibliography	131

Chapter 1

INTRODUCTION

Digital communication systems are used for transferring information data between separate remote points, which are connected by appropriate communication channels. The data being sent is usually affected by different channel errors (e.g., noise or interference) that worsen the quality of transmission link. The communication system performance and quality of transmissions can be increased by applying the *Forward Error Correction (FEC)* technique, which is used in almost all digital communication systems to improve performance with regards to Bit Error Rate (BER). Theoretically, FEC allows the maximum level of information in any channel. In practice, it reduces the cost for designing the communication system.

This is also the case for *Asymmetric Digital Subscriber Line (ADSL)*, which uses both *Trellis coding* and *Reed-Solomon FEC* to perform error corrections. In ADSL, the error correction algorithms used (especially decoding) are computationally complex and demanding compared to the other algorithms used (e.g., encoding, modulation, etc). Thus, to meet the given time constraints and to save hardware resources, an efficient implementation is preferred. With increasingly strict time-to-market, a fast process of implementation is also required.

In order to realize fast and efficient implementations, the use of different design strategies/methodologies becomes very important. Without having a strictly defined design strategy, many development projects fail to create effective systems on time and within available budget. However, to develop an expedient design strategy, even both academics and commercial firms that specialized in devising such designs spend tremendous effort for that.

1.1. Project Objective

The main objective of the project is to investigate whether a *fast* design strategy for an *efficient* implementation of a Reed-Solomon (RS) decoder specified in ITU G.992.1 [1] (is the standard for ADSL) on the given target architectures (i.e., DSP and FPGA) can be provided or not. In order to accomplish this, the proposed design trajectory, described in Section 1.3, is evaluated. The design trajectory is based on the *A³ framework*. The main idea of this framework is presented in the next section.

1.2. A^3 Framework

The typical A^3 framework consists of three related domains: *Applications*, *Algorithms* and *Architectures* (Figure 1.2.1). They are described below:

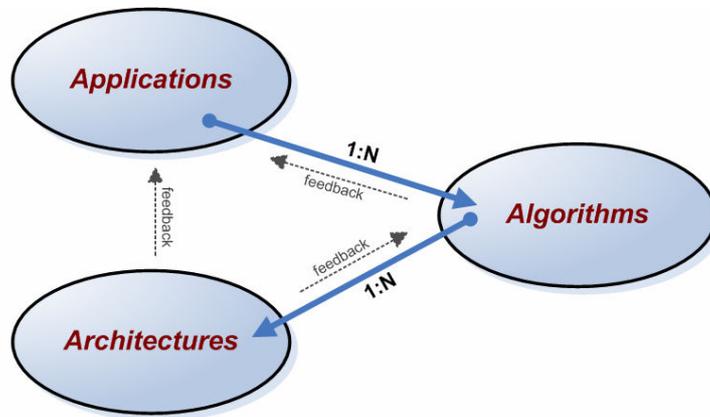


Figure 1.2.1: Typical A^3 framework.

- **Applications domain (first step):**

This domain is used for:

1. Specifying the system;
2. System analysis;
3. Defining the main tasks that the system must perform.

As we see in Figure 1.2.1, this domain has the relation with the “Algorithms” domain. This relation is “one-to-many” (1:N). It means that there may be a lot of algorithms in the “Algorithms” domain that can be used for a mathematical description of the system functionality. However, from existing algorithms we need to select only one, which best satisfies the given application requirements.

- **Algorithms domain (second step):**

This domain is mainly used for algorithms development, simulation and selection. From the simulation results we are able to see, which algorithm is more appropriate for the given application. Although sometimes it is difficult to find an optimal algorithm for the given application, and so the dashed line from the “Algorithms” domain to the “Applications” domain (see Figure 1.2.1) indicates that if such problem occurs, it is necessary to review the application and maybe it is useful to make some changes in the given specification to achieve better results.

As shown in Figure 1.2.1, the “Algorithms” domain has the relation with the “Architectures” domain. This relation is “one-to-many” (1:N). It means that the chosen algorithm can be implemented on different types of architectures (e.g. GPP, DSP, FPGA, etc.), or different architectures of a particular type.

- **Architectures domain (third step):**

In the “Architectures” domain the selected algorithm is implemented on the given architecture(s). As we see in Figure 1.2.1, this domain has the two dashed lines: one line is pointed to the “Algorithms” domain, meaning that not always the best algorithm for the given application will show the best result in a certain architecture, and so, in such case, we need to return one step back to the “Algorithms” domain for another algorithm selection. Other dashed line is pointed to the “Applications” domain, meaning that when we map an algorithm to the target architecture(s), the result of this mapping sometimes may not fulfill the given application requirements. In such case, it is necessary to review and possibly change these requirements.

1.3. Design Trajectory

The proposed design trajectory, shown in Figure 1.3.1, slightly differs from the typical A^3 framework, illustrated in Figure 1.2.1. Each particular domain of the proposed design trajectory is described below:

- **Applications domain:**

In this domain the ADSL technology with RS coding (i.e., encoding/decoding) are analyzed as the application of the project.

- **Algorithms domain:**

In this domain, first of all, the appropriate algorithms for RS coding are described in a structured way. In addition, the simulation of described algorithms is run to verify the functionality and evaluate the performance of the corresponding system. According to the limitations of the project (described in Section 1.4), the two different RS decoding algorithms are considered for their further analysis and implementation on the target architectures.

For the reason that the implementation may be performed on different types of architectures (i.e., DSP and FPGA), the following question occurs: which type of architecture is the most suitable for the execution of a certain algorithm? In order to answer this question, *algorithm characterization* is an option. The main idea of characterization is to extract relevant information from the specification of an application (i.e., algorithm) to guide the designer towards an efficient algorithm-architecture matching. For this purpose, different metrics can be efficiently used to rapidly stress the proper architecture style for the given application. In our case, this is referred to as a fast implementation.

- **Architectures domain:**

After the characterization of the given RS decoding algorithms, the implementation is performed. In order to implement the decoding algorithms, the two available devices

are used in the project: *Xilinx Virtex II* reconfigurable FPGA (v3000) and *TigerSHARC ADSP-TS201* DSP. The functionalities of these devices are presented in Appendixes A and B, respectively.

To verify that the obtained characterization results are true, the two decoding algorithms are first *optimized* (considering the capabilities of the target architectures) and then implemented both onto FPGA and DSP, resulting in four outputs: two architectures with two different algorithms in each (see Figure 1.3.1). For the desired verification, the corresponding implementation results are then compared with the characterization results.

Finally, only one particular architecture with one particular algorithm, which best satisfy the defined *cost function* related to the ADSL requirements, is selected from existing implementation outputs.

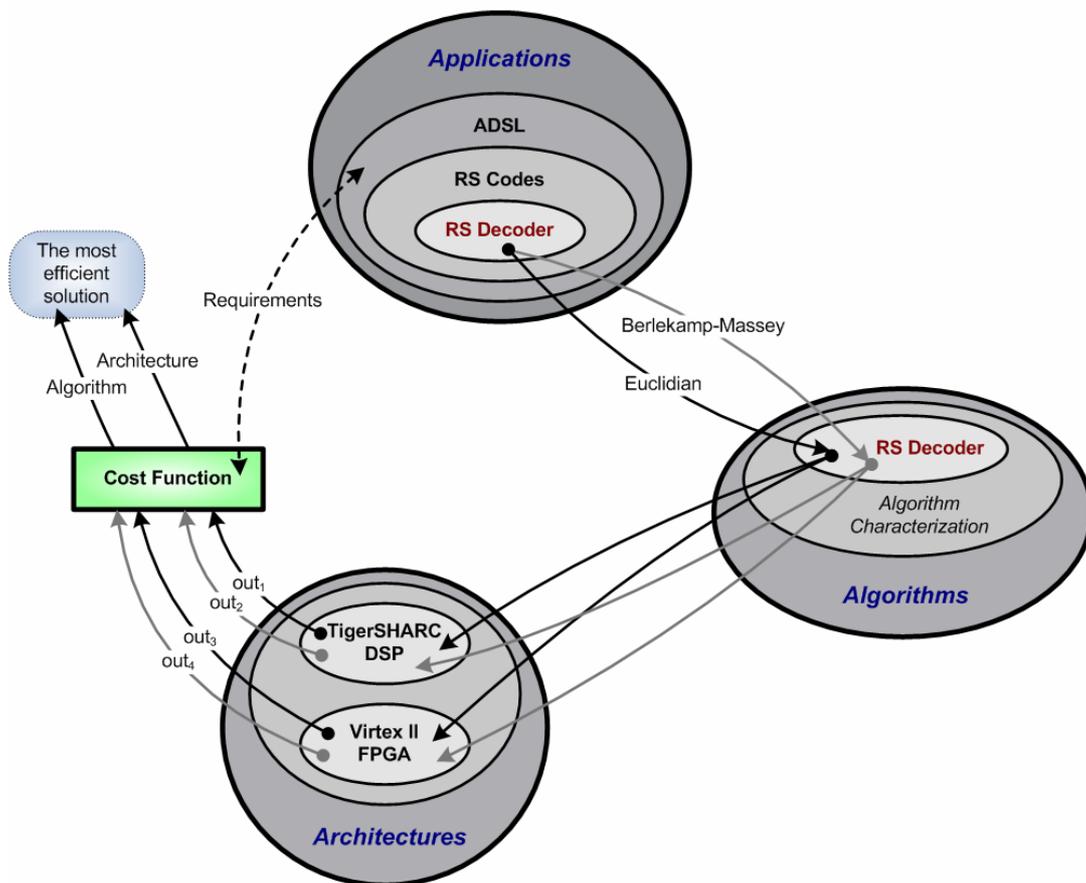


Figure 1.3.1: A^3 framework of the project.

1.4. Project Limitations and System Constraints

Because of the limited project period, a number of constraints have been made:

- It is decided to focus on the two commonly used RS decoding algorithms: *Berlekamp-Massey* and *Euclidean* algorithms (described in Chapter 3);
- The *erasure technique* (described in Chapter 3) in RS decoding is not considered;
- The programs for simulation and implementation of RS codes are built upon existing ones¹, written in C/C++ language;
- RS codes have a number of parameters, which can be modified. For the easier code optimization and implementation purposes, these parameters should be kept fixed.

1.5. System Requirements

There are several requirements for the system (i.e., RS decoder) to be developed during the project period:

- The system should follow ITU G.992.1 [1], which defines the system to support a minimum of 6.144 Mbit/s downstream;
- Changing the RS parameters, bit-error performance of RS codes changes as well. Since these parameters are selected to be constant in our case, we need to initially make a decision on the level of bit-error performance, which will correspond to a particular set of RS parameters. In order to feel the power of RS codes, it is decided to extract the highest available bit-error performance from RS codes and apply this performance to the system.

1.6. Organization of the Report

The report is divided into three parts. The first part consists of Chapters 2 and 3, where the ADSL technology with RS coding (i.e., encoding/decoding) are described as the application of the project. The second part is related to the “Algorithms” domain in the proposed design trajectory (Figure 1.3.1), and consists of Chapter 4, where the simulation is performed, and Chapter 5 (“Algorithm Characterization”). The last part of the report is related to the “Architectures” domain in the proposed design trajectory, and presented in Chapter 6 (“FPGA Implementation”), Chapter 7 (“DSP Implementation”), and Chapter 8, which carries out the verification of the algorithm characterization results.

¹ <http://www.eccpage.com/>

Chapter 2

ADSL SYSTEM

In recent years, Digital Subscriber Line (DSL) technology has been gaining popularity as a high speed network access technology, capable of the delivery of multimedia services over the existing telephone infrastructure. A major impairment for DSL is *impulse noise* in the telephone line. Lightning and switching equipment transients are common causes of impulse noise. However, current DSL services make use of forward error correction (FEC) techniques for improved resistance to noise interference in the data transmission.

The current chapter explains the FEC mechanism and presents the background to Asymmetric DSL (ADSL) system, which is a particular version of DSL technology.

2.1. Forward Error Correction in Transceivers

Forward error correction (FEC) in *transceiver* (transmitter/receiver pair) is used to deliver information from a source (transmitter) to a destination (receiver) through a noisy communication channel with a minimum of errors. FEC allows a receiver in the system to perform Error Detection and Correction (EDAC) without requesting a retransmission of the corrupted data. FEC offers a number of benefits:

- FEC enables a system to achieve high data reliability;
- FEC results in greater effective throughput of user data, because valuable bandwidth is not being used to retransmit corrupted data;
- FEC yields performance gains and low error rates for systems in which other options, such as increasing the transmitted power or installing noise-limiting components, are too expensive or impractical;
- System costs can be reduced by eliminating an expensive or sensitive component and compensating for the lost performance by a suitable FEC scheme.

Figure 2.1.1 depicts a typical FEC communication scheme:

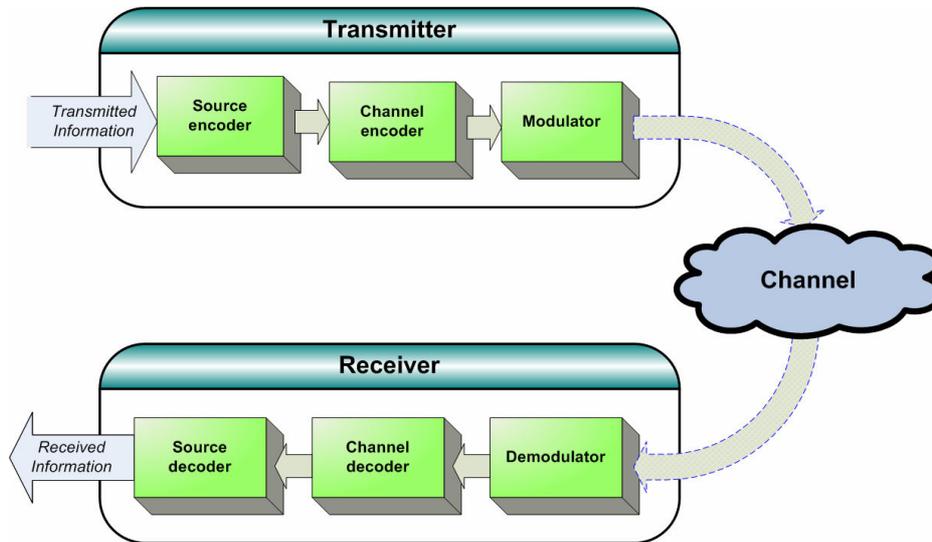


Figure 2.1.1: FEC communication system.

- **Source encoder:**

In the *source encoder* of the transmitter (Figure 2.1.1), the message to be transmitted is transformed into a sequence of bits that represents the original message. These bits are then fed to the *channel encoder*.

- **Channel encoder:**

The channel encoder (or FEC encoder) is designed to perform error correction with the aim of converting an unreliable communication channel into a reliable one. The encoder adds *redundancy* to the data produced by the source encoder in the form of *parity information*. Then at the receiver, a *channel decoder* is able to exploit the redundancy in such a way that a reasonable number of errors introduced by the channel can be corrected. Without redundancy, the code would not allow us to detect the presence of errors and therefore would not have any error controlling properties.

- **Modulator:**

The coded data produced by the channel encoder is then mapped into analogue signal (waveforms) in the *digital modulator*, and fed to the channel.

- **Channel:**

The channel provides the communication link between the transmitter and receiver, and introduces various forms of corruption to the transmitted signal, like environment noise, attenuation, etc. The errors introduced by the communication channel are classified into two main categories:

- *Random errors*. The probability of error is independent from one transmitting symbol to the next. Random errors occur in the *Additive White Gaussian Noise (AWGN)* channel in which the transmitted signal suffers the addition of wide-band noise whose amplitude is a normally (Gaussian) distributed random variable;
- *Burst errors*. The bit errors occur sequentially in very short time as groups. For example, impulse noise can cause a burst of errors. Impulse noise is a short burst of relatively high energy noise.

Thus, the task of the receiver is to capture the transmitted signal, and remove the effects of the channel.

- **Demodulator:**

The *demodulator* converts the waveforms received from the channel into a binary sequence, which is fed to the channel decoder.

- **Channel decoder:**

The job of the channel decoder (or FEC decoder) is to decide what the transmitted information was. The channel decoder removes the redundancy introduced by the channel encoder in the transmitter, and attempts to detect and correct possible bit errors using the knowledge of the code used by the channel encoder and the redundancy contained in the received data. The frequency at which bit errors occur at the output of the channel decoder is a measure of the demodulator-decoder performance. Typically the bit error rate (BER) at this point is kept at a desired level so as to have acceptable quality of communication with minimum resource usage.

- **Source decoder:**

Finally, the *source decoder* tries to reconstruct the original message from the decoded data. This will be an estimation of the original message due to the possible corruption introduced to the data along its way through the communication link.

2.1.1. Error-Control Codes

FEC is also known as channel coding (realized by the FEC encoder/decoder), which is based on a specific error-control code. There are the two main types of error-control codes used in communication systems:

1. *Block codes*. Block codes are based strictly on finite field arithmetic. They can be used to either detect or correct errors;
2. *Convolutional codes*. These codes are developed with a separate strong mathematical structure and are primarily used for real-time error correction.

The question of whether to choose block codes or convolutional codes depends on the following. When the environment consists predominately of random errors, convolutional codes provide a low bit error rate (BER) solution. However, when the environment consists mainly of burst errors, block codes often perform even better.

Some applications, such as ADSL (described in the following section), use both convolutional and block codes. In such case, concatenated codes result in strong performance by operating in two steps.

2.2. Overview of ADSL System

Digital Subscriber Line (DSL) technology is a home user-oriented modem technology that uses existing twisted-pair copper telephone lines to transport high-bandwidth data, such as multimedia and video. The technology is attractive in the aspect that it utilizes the telephone system infrastructures, usually already installed in buildings and facilities. In the Plain Old Telephone System (POTS), only a fraction of the bandwidth of the copper loop (telephone line) is used, thus the DSL service is designed to use the excess bandwidth for downstream and upstream data transmission. DSL service is dedicated, point-to-point, public network access over twisted-pair copper wire on the local loop between a Network Service Provider (NSP's) central office and the customer site.

Some other popular services, such as a standard dial-up modem or an ISDN line, also use the telephone lines to communicate. However, those services prevent the simultaneous operation of standard analog phone service on the same phone line. An important advantage of DSL is that it allows the POTS signal to co-exist with the DSL data signal. The POTS channel is split off from the digital modem by filters commonly called “splitters”, thus guaranteeing uninterrupted POTS.

Asymmetric Digital Subscriber Line (ADSL) is the most widely used DSL standard today. The term asymmetric reflects the difference between upstream and downstream bit rates in the transmission link. ADSL allows more bandwidth downstream – from an NSP's Central Office (CO) to the customer site – than upstream from the subscriber to the central office. This asymmetry, combined with always-on access, makes ADSL ideal for Internet surfing, since users typically download much more information than they send.

ANSI standard T1.413 defines an ADSL system to transmit downstream and upstream data rates up to 6.8 Mbit/s and 640 kbit/s, respectively. ITU Recommendation G.992.1 [1] (is the standard for ADSL) defines a system based on T1.413 as a core, but expanded via three annexes to meet particular regional needs. The maximum data rates mentioned in the literature are about 8 Mbit/s downstream and 1 Mbit/s upstream, see [1] or [2].

2.2.1. Spectrum Allocation

According to [1] (annex A), ADSL is designed to provide data transmission on loops up to 5 km over a 25 kHz – 1.1 MHz frequency band. An ADSL circuit connects an

ADSL modem on each end of a twisted-pair telephone line, creating three information channels – a downstream channel, an upstream channel, and a basic telephone service channel. Each of these channels has its own frequency band. The POTS band goes from near DC to approximately 4 kHz. A frequency guard band is placed between the POTS spectrum and the ADSL spectrum to help avoid interference. The ADSL spectrum (downstream and upstream bands) starts above the POTS band and extends up to approximately 1.1 MHz. There are actually two different ways that the ADSL spectrum can be arranged: to create multiple channels, ADSL modems divide the available bandwidth of a telephone line in one of two ways: *Frequency-Division Multiplexing (FDM)* or *echo-cancellation*, as shown in (Figure 2.2.1): [3]

- **Frequency-Division Multiplexing (FDM):**

In FDM mode, the upstream and downstream frequency bands are separated. Using FDM, the upstream channel allocation ranges from about 26 kHz to 138 kHz, and downstream ranges from 138 kHz to 1.1 MHz (Figure 2.2.1a).

- **Echo-cancellation:**

An alternative to FDM is to use echo-cancellation, which enables upstream and downstream signals to use the same spectrum (Figure 2.2.1b). The objective of echo-cancellation is to enable the downstream data to use lower frequencies than are available in FDM mode. Using lower frequencies, we can achieve less signal attenuation, which theoretically allows faster downstream data rates on longer loops [2]. Echo-cancellation also adds more available spectrum to the downstream channel.

However, it does require echo-canceling circuitry to remove the reflection of the locally transmitted signal in the overlapped band. This certainly increases the complexity of digital signal processing in the receivers.

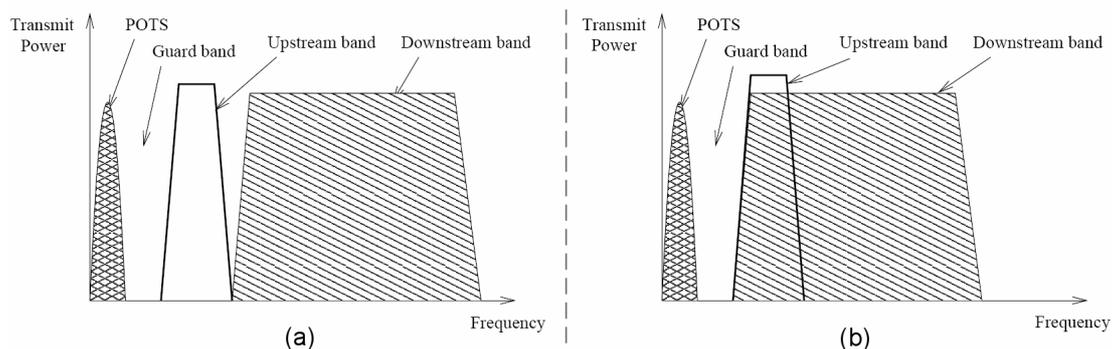


Figure 2.2.1: Frequency spectrum usage by ADSL with: (a) FDM, and (b) echo-cancellation.

2.2.2. ADSL Modem

Each ADSL modem consists of transmitter and receiver. A block diagram of a typical ADSL transmitter/receiver pair (transceiver) is shown in Figure 2.2.2.

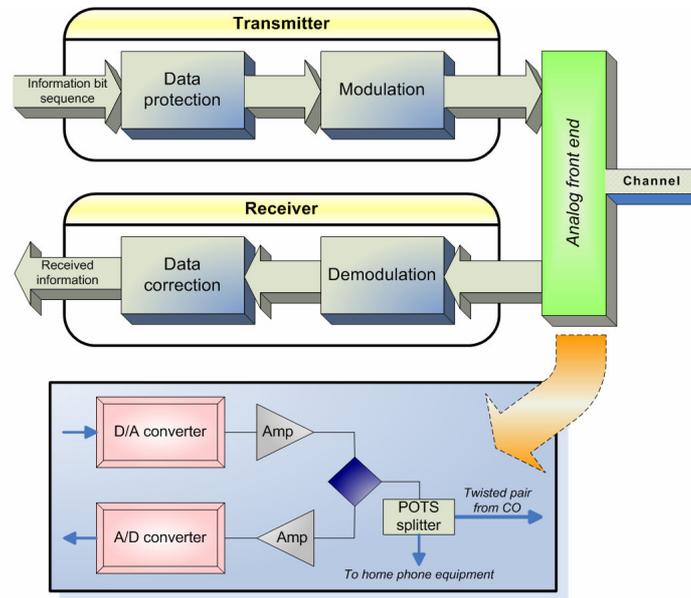


Figure 2.2.2: Block diagram of a typical ADSL modem.

At the transmitter the information data to be sent is first protected, then modulated and finally transmitted. At the receiver the obtained data is first demodulated and then corrected from errors introduced by the communication channel. Each of these steps is briefly described in the following sections. The analog front end (Figure 2.2.2) is of no interest in our case, and its description is therefore beyond the scope of this report.

2.2.3. Data Protection and Correction

The physical layer of ADSL must ensure that data is transferred reliably across the channel, so the process of data protection is performed: *Cyclic Redundancy Check (CRC)* attachment, data coding (FEC technique) and *interleaving* are designed to provide this (Figure 2.2.3).

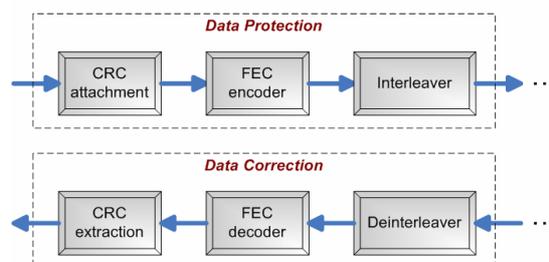


Figure 2.2.3: Data protection and correction blocks of an ADSL modem, shown in Figure 2.2.2.

2.2.3.1. Cyclic Redundancy Check (CRC)

CRC is a method used to detect errors in the received signal. This is done by converting the binary signal to a polynomial and then dividing it with a predefined polynomial called the key. The remainder in this division is called CRC. The signal together with the CRC is transmitted. The receiver performs the same operation as the transmitter, dividing the signal with the same predefined polynomial (key), and then checks the difference between the obtained remainder (CRC) at the receiver and CRC received from the transmitter. If the difference is zero, there is a high probability that the signal has been received correctly, otherwise an error has probably occurred. [4]

2.2.3.2. Forward Error Correction (FEC)

Reed-Solomon (RS) codes (are block codes) have been chosen for the FEC technique in ADSL [1]. The RS encoder takes k data symbols of 8-bits each (byte) and adds parity symbols (redundancy) to make an n symbol data block, called *codeword*. The maximum length (starting from $n = 1$) of a codeword with 8-bit symbols in ADSL is 255 bytes. There are $(n - k)$ redundant bytes. The ADSL standard requires support of all even numbers from 0 to 16 of redundancy bytes per codeword. This would allow for up to 8 bytes to be in error for every RS codeword.

The essence of RS codes and the principles of RS encoding and decoding are presented in detail in the next chapter.

2.2.3.3. Interleaving

The purpose of the combination of an interleaver in the transmitter and a deinterleaver in the receiver is to spread burst of errors, which occur between several (usually the two), over many codewords, and thus reduce the number of errors in any one codeword to what can be corrected by the Reed-Solomon decoder. The two important parameters for the interleaver are the number of bytes per codeword, n , and the *interleave depth*, D . An interleaver of depth D reads D codewords of length n each and arranges them in a block (array) with D rows and n columns. Then the codewords in the formed array are *convolutionally interleaved* (see [2]) and fed to the channel. In the deinterleaver the bits are rearranged back to its original order. ADSL supports interleave depth which is a power of two from 1 to 64.

The higher interleave depth is, the more data can be interleaved, resulting in much more effective RS FEC performance. But increasing the interleave depth will cause additional latency or delay in the time the data is transmitted and the time it is available to the receiving user.

2.2.3.4. Fast and Interleaved Paths

There are actually two separate paths in the data protection and correction blocks of an ADSL modem: “*fast*” and *interleaved*. In the “fast” path the interleaving is not used. The interleaved path provides a lower error rate, but higher latency in comparison with the non-interleaved “fast” path. The increased latency normally

causes no problems for general data transmission, but digitized voice over a high-latency path results in extremely unpleasant echo. For this reason, a minimum interleave depth (or no interleaving) is always used on data channels carrying voice traffic. As delay is added to voice transmissions, the problem of echo increases radically and requires additional treatment.

Deciding on a compromise between burst error rate and latency for each data channel is a function of the Transmission Convergence (TC) layer in ADSL [2], which must combine the multiple input data channels and assign them to either the “fast” or the interleaved path.

2.2.4. Modulation

The physical layer of ADSL uses a multicarrier modulation technique, known as *Discrete Multi-Tone (DMT)*, to create the ADSL signal. The basic idea of DMT is to split the available bandwidth into a large number of subchannels, where each subchannel uses *Quadrature Amplitude Modulation (QAM)* [5].

With ADSL, the frequency band in DMT is divided into N narrowband channels (subchannels) of about $\Delta f = 4$ kHz each for transmission, where each subchannel may be approximated by a flat transfer function $|H|$, as illustrated in Figure 2.2.4 [6]. In the downstream direction the maximum number of subchannels is $N = 255$, which are placed at frequencies $n\Delta f$, $n = 1$ to 255. In the upstream direction the maximum number of subchannels is $N = 31$, placed at $n\Delta f$, $n = 1$ to 31.

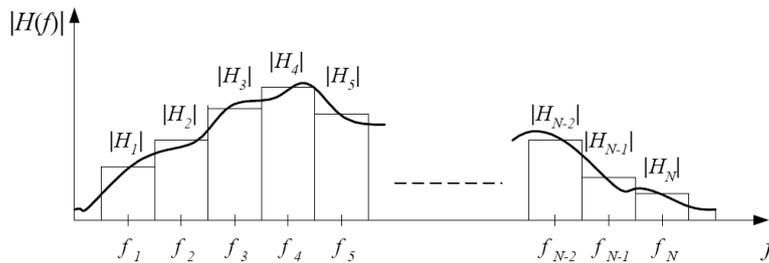


Figure 2.2.4: An example of subchannels response.

A DMT system transmits data in parallel over several narrowband channels. DMT is able to allocate data so that the throughput of every single subchannel is maximized by sending different numbers of bits on different subchannels. The number of bits on each subchannel depends on the Signal-to-Noise Ratio (SNR) of the corresponding subchannel, and this is referred to as *bit-loading* (described in Section 2.2.4.2).

The DMT signal is formed by using an Inverse Fast Fourier Transform (IFFT) to generate orthogonal subchannels (don’t interfere with each other) at the transmitter. The data symbols at the transmitter are treated as being in the frequency domain and act as complex weights for the basis functions (orthogonal sinusoids at different frequencies) of the IFFT. The IFFT then converts the data symbols into a time-domain “sum of sinusoids” signal. The block of IFFT output samples is known as a DMT symbol. This time-domain signal is transmitted across the channel, and an FFT is used at the receiver to bring the signal back into the frequency domain.

A simplified diagram of an ADSL modulation block is shown in Figure 2.2.5.

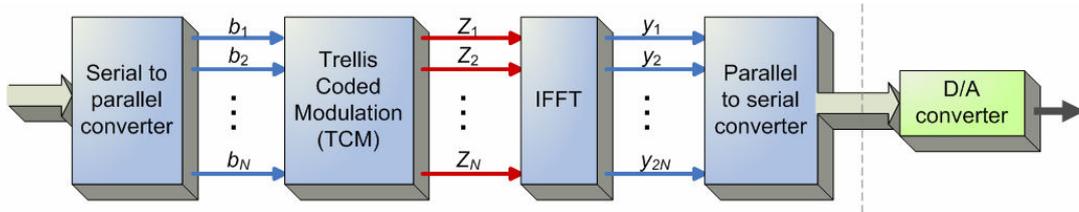


Figure 2.2.5: A simplified diagram of the modulation block of an ADSL modem, shown in Figure 2.2.2.

The DMT modulation consists in dividing the consecutive data into blocks and encoding them with *Trellis Coded Modulation (TCM)* into a set of N multibit complex symbols Z_i (Figure 2.2.5). IFFT is then applied on the set of complex symbols. In the end, $2N$ real samples are generated and passed through a Digital-to-Analogue (D/A) converter. [2]

2.2.4.1. Trellis Coded Modulation (TCM)

TCM, shown in Figure 2.2.5, is a technique combining Trellis (convolutional) coding and QAM modulation in a single operation. In particular, Trellis coding in TCM is a process of altering the QAM constellation to provide better performance in a noisy environment. TCM is a bandwidth efficient scheme, where the redundancy introduced by the coding does not expand the bandwidth. This allows reliable high data-rate communication over channels with limited bandwidth.

The Trellis decoding is based on the Viterbi algorithm. Trellis coding together with Viterbi decoding are typically designed to reduce errors from AWGN. However, the nature of the decoding algorithm is such that the decoder can cause burst errors to occur if errors are made during the decoding process. Moreover, Trellis coding requires more complex transceivers, and Trellis capable chipsets may have a slightly higher internal power requirement.

The ADSL standard gives as an option the possibility to Trellis code the modulation. Thus, Reed-Solomon (RS) and Trellis coding can be combined in a concatenated coding scheme (i.e., block codes + convolutional codes), resulting in strong performance by operating in two steps. In the concatenated coding scheme, RS is the *outer code*, and Trellis code is the *inner code*. The information is first encoded by the outer code, and then the encoded sequence is further encoded by the inner code. RS codes in ADSL are used as outer codes because of their ability to correct the burst errors from the inner decoder.

2.2.4.2. Bit-loading

In a DMT system the subchannels carry different number of bits depending on their respective SNR. This is referred to as bit-loading. Since the channel is stationary, the bit-loading factors are calculated in an initial ADSL training session. During initial training, the ADSL modem tests which of the available subchannels have an

acceptable SNR. If SNR is low, the corresponding noisy regions of the spectrum can be “loaded” with fewer bits. If SNR is not satisfied, the corresponding subchannels will not be used altogether, merely resulting in reduced throughput on an otherwise functional ADSL connection [2]. The SNR of subchannel j can be calculated as

$$SNR_j = \frac{E_j |H(f_j)|^2}{\sigma_j^2} \quad (2.2.1)$$

where E_j is the average signal energy on subchannel j , $|H(f_j)|$ is the transfer function of subchannel j in the sampled frequency f_j (see Figure 2.2.4), and σ_j^2 is the noise variance.

2.3. Summary

This chapter explained the main concept of the FEC mechanism, and briefly described each block of the FEC system. Moreover, this chapter presented the background to ADSL system, which is a particular version of DSL technology. With ADSL, the functionality of each block of an ADSL modem was explained.

Chapter 3

REED-SOLOMON CODES

The current chapter summarizes the essence of the Reed-Solomon (RS) codes and provides background information on finite field. Furthermore, this chapter introduces the general concept of finite field arithmetic implementation. Finally, the principles of RS encoding and decoding are explained.

In order to compose this chapter, the following literature was mainly used: [7], [8] and [9].

3.1. Introduction to Reed-Solomon Codes

RS codes are error detection and correction (EDAC) scheme used in different forward error correction (FEC) techniques. These codes provide powerful correction, have high channel efficiency, and thus have a wide range of applications in digital communications and storage, e.g.:

- Storage devices: Compact Disk (CD), DVD, etc;
- Wireless or mobile communications: cellular phones, microwave links, etc;
- Satellite communications;
- Digital television / DVB;
- High-speed modems: ADSL, VDSL, etc.

As we will see later in this chapter, RS codes are particularly well suited for correcting burst errors. They are based on a special area of mathematics known as finite fields.

3.2. Properties of Reed-Solomon Codes

RS codes are linear block codes. A RS code is specified as $RS(n, k)$ with m -bit symbols. $RS(n, k)$ codes on m -bit symbols exist for all n and k for which

$$0 < k < n < 2^m + 2 \quad (3.2.1)$$

where k is the number of data symbols being encoded, and n is the total number of code symbols in the encoded block, called *codeword*. This means that the RS encoder takes k data symbols of m -bits each and adds parity symbols (redundancy) to make an n symbol codeword. There are $(n - k)$ parity symbols of m -bits each. For the most conventional RS(n, k) code,

$$(n, k) = (2^m - 1, (2^m - 1) - 2t) \quad (3.2.2)$$

where t is the *symbol-error correcting capability* of the code, and $(n - k) = 2t$ is the number of parity symbols. It means that the RS decoder can correct up to t symbols that contain errors in a codeword, that is, the code is capable of correcting any combination of t or fewer errors, where t can be expressed as

$$t = \left\lfloor \frac{n-k}{2} \right\rfloor \quad (3.2.3)$$

Equation (3.2.3) illustrates that for the case of RS codes, correcting t symbol errors requires no more than $2t$ parity symbols. For each error, one redundant symbol is used to locate the error in a codeword, and another redundant symbol is used to find its correct value. Denoting the number of errors with an unknown location as n_{errors} and the number of errors with known locations (*erasures*) as $n_{erasures}$, the RS algorithm guarantees to correct a codeword, provided that the following is true

$$2n_{errors} + n_{erasures} \leq 2t \quad (3.2.4)$$

Expression (3.2.4) is called simultaneous error-correction and erasure-correction capability. Erasure information can often be supplied by the demodulator in a digital communication system. Nevertheless, the erasure technique is an additional feature that is sometimes incorporated into decoders for RS codes and requires separate handling. Thus, according to the system constraints, described in Section 1.4, we do not deal with erasures, and consider only error correction.

Keeping the same symbol size m , RS codes may be *shortened* by (conceptually) making a number of data symbols zero at the encoder, not transmitting them, and then re-inserting them at the decoder. For example, the RS(255, 223) code ($m = 8$) can be shortened to RS(200, 168) with the same $m = 8$. The encoder takes a block of 168 data bytes, (conceptually) adds 55 zero bytes, creates a RS(255, 223) codeword and transmits only the 168 data bytes and 32 parity bytes.

3.2.1. Reed-Solomon Codes Perform Well Against Burst Noise

Consider a popular Reed-Solomon code RS(255, 223), where each symbol is made up of $m = 8$ bits (such symbols are referred to as *bytes*). Since $(n - k) = 32$, Equation (3.2.3) indicates that this code can correct any 16 symbol errors in a codeword of 255

bytes. Now assume the presence of a noise burst, lasting for 128-bit durations and disturbing one codeword during transmission, as illustrated in Figure 3.2.1.

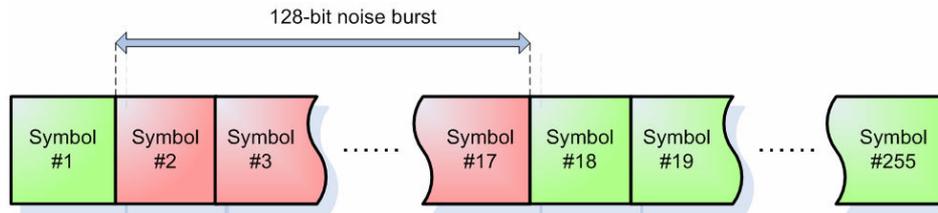


Figure 3.2.1: A codeword of 255 bytes disturbed by 128-bit noise burst.

In this example, a burst of noise that lasts for a duration of 128 contiguous bits corrupts exactly 16 symbols. The RS decoder for the (255, 223) code will correct any 16 symbol errors without regard to the type of damage suffered by the symbol. In other words, when a decoder corrects a byte, it replaces the incorrect byte with the correct one, whether the error was caused by one bit being corrupted or all eight bits being corrupted. Thus if a symbol is wrong, it might as well be wrong in all of its bit positions. That is why RS codes are extremely popular because of their capacity to correct burst errors.

3.3. Galois Fields

The algorithms for RS encoding and decoding require algebraic operations over finite fields in which a polynomial is used to represent data sequences. Thus, in order to understand the encoding and decoding principles of RS codes, first of all it is necessary to venture into the area of finite fields known as *Galois Fields (GF)*, since these codes are based on the use of Galois field arithmetic.

3.3.1. Properties of Finite Field

The formal properties of a finite field, which has a finite number of elements, are: [7]

- There are two defined operations, namely addition and multiplication;
- The result of adding or multiplying two elements from the field is always an element in the field;
- One element of the field is the element *zero*, such that $a + 0 = a$ for any element a in the field;
- One element of the field is *unity*, such that $a \cdot 1 = a$ for any element a in the field;
- For every element a in the field, there is an additive inverse element $-a$, such that $a + (-a) = 0$. This allows the operation of subtraction to be defined as addition of the inverse;

- For every non-zero element b in the field there is a multiplicative inverse element b^{-1} , such that $bb^{-1} = 1$. This allows the operation of division to be defined as multiplication by the inverse;
- Both an addition and a multiplication operation that satisfy the commutative, associative, and distributive laws.

These properties can be only satisfied if the field size is any *prime number* or any integer power of a prime.

3.3.2. Prime Size Finite Field $GF(p)$

For any prime number, p , there exists a finite field denoted $GF(p)$ that contains p elements. The rules for a finite field with a prime number p of elements can be satisfied by carrying out the arithmetic modulo- p .

In any prime size field, it can be proved that there is always at least one element whose powers constitute all the non-zero elements of the field. This element is said to be *primitive*. For example, in the field $GF(7)$, the number 3 is primitive as

$$3^0 = 1, \quad 3^1 = 3, \quad 3^2 = 2, \quad 3^3 = 6, \quad 3^4 = 4, \quad 3^5 = 5$$

Higher powers of 3 just repeat the pattern as $3^6 = 1$, and so on.

3.3.2.1. Binary Field $GF(2)$

The simplest Galois field is $GF(2)$, where $p = 2$. Its elements are the set $\{0, 1\}$ under modulo-2 algebra. The addition and multiplication tables of $GF(2)$ are shown in Tables 3.3.1 and 3.3.2.

+	0	1
0	0	1
1	1	0

Table 3.3.1: Modulo-2 addition (XOR operation).

×	0	1
0	0	0
1	0	1

Table 3.3.2: Modulo-2 multiplication.

There is a one-to-one correspondence between any binary number and a polynomial in that every binary number can be represented as a polynomial over $GF(2)$, and conversely. A polynomial of degree D over $GF(2)$ has the following general form:

$$f(X) = f_0 + f_1X + f_2X^2 + f_3X^3 + \dots + f_DX^D \quad (3.3.1)$$

where the coefficients f_0, \dots, f_D are the elements of $GF(2)$. A binary number of $(N + 1)$ bits can be represented as an abstract polynomial of degree N by taking the coefficients equal to the bits and the exponents of X equal to the bit locations. Thus, in the polynomial representation, a multiplication by X represents a shift to the left,

i.e. to one position earlier in the sequence. For example, the binary number 10011 is equivalent to the following polynomial:

$$10011 \leftrightarrow 1 + X + X^4$$

The bit at the zero position (the coefficient of X^0) is equal to 1, the bit at the first position (the coefficient of X) is equal to 1, the bit at the second position (the coefficient of X^2) is equal to 0, and so on.

3.3.3. Extensions to the Binary Field – $GF(2^m)$

As was mentioned, finite fields can also be created where the number of elements is an integer power of any prime number p .

Let us suppose that we wish to create a finite field $GF(q)$ and that we are going to take a primitive element of the field and assign the symbol α to it. The powers of α , α^0 to α^{q-2} , $(q - 1)$ terms in all, form all the non-zero elements of the field. The element α^{q-1} will be equal to α^0 , and higher powers of α will merely repeat the lower powers found in the finite field. In order to know how to add the powers of alpha, the best to understand this is to examine the case, where $q = 2^m$ (m is an integer).

For the field $GF(2^m)$ we know that

$$\alpha^{2^m-1} = \alpha^0 = 1$$

Since in $GF(2^m)$ algebra, plus (+) and minus (–) are the same, the last one can be represented as follows:

$$\alpha^{2^m-1} + 1 = 0$$

This will be satisfied if any of the factors of this polynomial are equal to zero. The factor that we choose here should be irreducible, and should not be a factor of $(\alpha^n + 1)$ for any value of n less than $(2^m - 1)$; otherwise, alpha will not be primitive. Any polynomial that satisfies these properties is called a *primitive polynomial*, and it can be shown that there will always be a primitive polynomial and thus there will always be a primitive element. Moreover, the degree of the primitive polynomials for $GF(2^m)$ is always m .

Now consider an example, where the field is $GF(2^3)$. The factors of $(\alpha^7 + 1)$ are

$$\alpha^7 + 1 = (\alpha + 1)(\alpha^3 + \alpha + 1)(\alpha^3 + \alpha^2 + 1)$$

Both the polynomials of degree 3 are primitive and so we choose, arbitrarily, the first, constructing the powers of a subject to the condition

$$\alpha^3 + \alpha + 1 = 0 \tag{3.3.2}$$

or

$$\alpha^3 = -1 - \alpha$$

Since in $\text{GF}(2^m)$, $+1 = -1$, α^3 can be represented as follows:

$$\alpha^3 = 1 + \alpha$$

So the other non-zero elements of the field are now found to be

$$\begin{aligned}\alpha^4 &= \alpha \cdot \alpha^3 = \alpha \cdot (1 + \alpha) = \alpha + \alpha^2 \\ \alpha^5 &= \alpha \cdot \alpha^4 = \alpha \cdot (\alpha + \alpha^2) = \alpha^2 + \alpha^3 = \alpha^2 + (1 + \alpha) = 1 + \alpha + \alpha^2 \\ \alpha^6 &= \alpha \cdot \alpha^5 = \alpha \cdot (1 + \alpha + \alpha^2) = \alpha + \alpha^2 + \alpha^3 = 1 + \alpha^2 \\ \alpha^7 &= \alpha \cdot \alpha^6 = \alpha \cdot (1 + \alpha^2) = \alpha + \alpha^3 = 1 = \alpha^0\end{aligned}$$

Note that $\alpha^7 = \alpha^0$, and therefore the eight finite field elements ($2^m = 2^3 = 8$) of $\text{GF}(2^3)$, generated by (3.3.2), are $\{0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6\}$. Here we notice that each new power of alpha is α times the previous power of alpha.

In general, extended Galois fields of class $\text{GF}(2^m)$ possess 2^m elements, where m is the symbol size, that is, the size of an element (in bits). For example, in ADSL systems, the Galois field is always $\text{GF}(2^8) = \text{GF}(256)$, where $m = 8$ (is fixed number). It is generated by the following primitive polynomial:

$$1 + X^2 + X^3 + X^4 + X^8 \quad (3.3.3)$$

Due to the one-to-one mapping that exists between polynomials over $\text{GF}(2)$ and binary numbers, the field elements of $\text{GF}(2^8)$ are representable as binary numbers of eight bits each, that is, as bytes. The following section illustrates this with $\text{GF}(2^3)$.

In order to implement $\text{GF}(2^m)$ in software or hardware, the field elements can be represented by the contents of a binary *Linear Feedback Shift Register (LFSR)* formed from a primitive polynomial, see [10] or [11]. In our case, α is set to 2 to generate the field elements of $\text{GF}(2^8)$ by means of (3.3.3).

3.3.4. Representation of Finite Field Elements

Let α be a primitive element of $\text{GF}(2^3)$ such that the primitive polynomial

$$p(\alpha) = \alpha^3 + \alpha + 1 = 0$$

The following table shows three different ways to represent elements in $\text{GF}(2^3)$:

Power	Polynomial	Vector $\alpha^2 \alpha^1 \alpha^0$
–	0	0 0 0
1	1	0 0 1
α	α	0 1 0
α^2	α^2	1 0 0
α^3	$1 + \alpha$	0 1 1
α^4	$\alpha + \alpha^2$	1 1 0
α^5	$1 + \alpha + \alpha^2$	1 1 1
α^6	$1 + \alpha^2$	1 0 1

Table 3.3.3: Different representations of $GF(2^3)$ elements.

The first column of Table 3.3.3 represents the powers of α . The second column shows the polynomial representation of the field elements. This polynomial representation was obtained in the previous section. And the last column of Table 3.3.3 is the vector representation of the field elements, where the coefficients of α^2 , α^1 and α^0 , taken from the second column, are represented as binary numbers. A one-to-one correspondence between any binary number and a polynomial was explained in Section 3.3.2.1.

Such representations of finite field elements are used to implement Galois field arithmetic. For example, when adding elements in $GF(2^m)$, the vector (binary) representation is the most useful, because a simple XOR operation is needed. However, when elements are going to be multiplied, the power representation is the most efficient. Using the power representation, a multiplication becomes simply an addition modulo $(2^m - 1)$. The polynomial representation may be appropriate when making operations modulo a polynomial.

3.3.5. $GF(2^m)$ Arithmetic Implementation

An opportune way to perform both additions and multiplications in $GF(2^m)$ is to use two look-up tables (*antilog* and *log*), with different interpretations of the address. This allows one to change between power representation and polynomial (vector) representation of an element of $GF(2^m)$.

The antilog table $A(i)$ is used when performing additions. The table gives the value of a binary vector, represented as an integer in natural representation, $A(i)$, that corresponds to the element α^i . The log table $L(i)$ is useful when performing multiplications. This table gives the value of a power of alpha, $\alpha^{L(i)}$, that corresponds to the binary vector represented by the integer i . The relation between $A(i)$ and $L(i)$ is expressed as:

$$\alpha^{L(i)} = A(i) \quad (3.3.4)$$

Now let's form the antilog and log tables from Table 3.3.3, where $\alpha^7 = 1$. The corresponding log and antilog tables are the following:

Element, α^i	Address, i	Antilog table, $A(i)$	Log table, $L(i)$
α^0	0	1	-1
α^1	1	2	0
α^2	2	4	1
α^3	3	3	3
α^4	4	6	2
α^5	5	7	6
α^6	6	5	4
-	7	0	5

Table 3.3.4: The log and antilog tables formed from Table 3.3.3.

With obtained tables, Galois field addition is easy to implement both in software and hardware, as it is the same as modulo-2 addition (XOR operation). Consider the computation of an element $\gamma = \alpha^3 + \alpha^5$. Using the antilog table, the computation of γ proceeds as follows:

$$A(3) \oplus A(5) = 3 \oplus 7 = (011)_2 \oplus (111)_2 = (100)_2 = 4, \text{ where } L(4) = 2 \Rightarrow \alpha^2$$

where \oplus is the XOR operation and $(\dots)_2$ represents a binary number. Besides, it is not difficult to perform multiplication. For instance, let's calculate the expression $\gamma = 2 \cdot 6$. The result is

$$\gamma = A(L(2) + L(6)) = A(1 + 4) = A(5) = 7$$

A zero element, which does not appear in the table, deserves special attention in the $\text{GF}(2^m)$ arithmetic implementation.

3.4. Reed-Solomon Encoding

The key to the RS encoding is to view the symbols of the message that is to be encoded as if they are the coefficients of a polynomial. As was described previously, when the RS encoder receives an information sequence, it creates encoded blocks consisting of $n = (2^m - 1)$ symbols each, where m is the symbol size in bits. The encoder divides the information bit sequence into message blocks of $k = (n - 2t)$ symbols. Each message block is equivalent to a message polynomial of degree $k - 1$, denoted as

$$m(X) = m_0 + m_1X + m_2X^2 + \dots + m_{k-1}X^{k-1}$$

where the coefficients m_0, m_1, \dots, m_{k-1} of the polynomial $m(X)$ are the symbols of a message block. Moreover, these coefficients are elements of $\text{GF}(2^m)$. So the information sequence is mapped into an abstract polynomial by setting the coefficients equal to the symbol values.

Consider the Galois field $\text{GF}(2^8)$, so the information sequence is divided into symbols of eight consecutive bits each (Figure 3.4.1). The first symbol in the sequence is 00000001. In the power representation, 00000001 becomes $\alpha^0 \in \text{GF}(2^8)$. Thus, α^0 becomes the coefficient of X^0 . The second symbol is 00000100, so the coefficient of X^1 is α^2 . The third symbol is 11111101, so the coefficient of X^2 is α^{80} and so on.

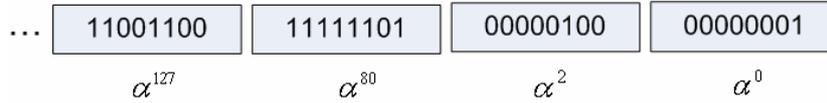


Figure 3.4.1: The information bit sequence divided into symbols.

The corresponding message polynomial is $m(X) = \alpha^0 + \alpha^2 X + \alpha^{80} X^2 + \dots$

3.4.1. Systematic Encoding

The encoding of RS codes is performed in systematic form. In systematic encoding, the encoded block (codeword) is formed by simply appending parity (or redundant) symbols to the end of the k -symbols message block, as shown in Figure 3.4.2. In particular, codeword's k -symbols message block consists of k consecutive coefficients of a message polynomial, and $2t$ parity symbols are the coefficients (from $\text{GF}(2^m)$) of a redundant polynomial.

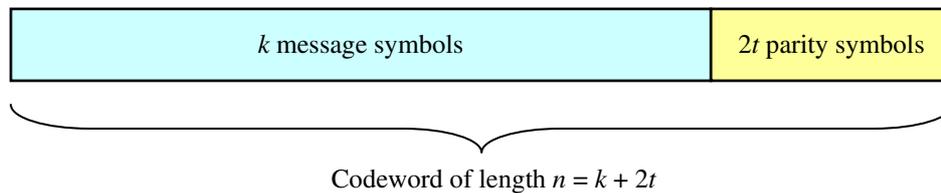


Figure 3.4.2: A codeword is formed from message and parity symbols.

Applying the polynomial notation, we can shift the information into the leftmost bits by multiplying by X^{2t} , leaving a codeword of the form

$$c(X) = X^{2t}m(X) + p(X) \quad (3.4.1)$$

where $c(X)$ is the codeword polynomial, $m(X)$ is message polynomial and $p(X)$ is the redundant polynomial.

The parity symbols are obtained from the redundant polynomial $p(X)$, which is the remainder obtained by dividing $X^{2t}m(X)$ by the *generator polynomial*, which is expressed as

$$p(X) = (X^{2t}m(X)) \bmod g(X) \quad (3.4.2)$$

So, a RS codeword is generated using a generator polynomial, which has such property that all *valid codewords* (i.e., not corrupted after transmission) are exactly divisible by the generator polynomial. The general form of the generator polynomial is:

$$\begin{aligned} g(X) &= (X + \alpha)(X + \alpha^2)(X + \alpha^3)\dots(X + \alpha^{2t}) \\ &= g_0 + g_1X + g_2X^2 + \dots + g_{2t-1}X^{2t-1} + X^{2t} \end{aligned} \quad (3.4.3)$$

where α is a primitive element in $\text{GF}(2^m)$, and $g_0, g_1, \dots, g_{2t-1}$ are the coefficients from $\text{GF}(2^m)$. The degree of the generator polynomial is equal to the number of parity symbols. Since the generator polynomial is of degree $2t$, there must be precisely $2t$ consecutive powers of α that are roots of this polynomial. We designate the roots of $g(X)$ as $\alpha, \alpha^2, \dots, \alpha^{2t}$. It is not necessary to start with the root α , because starting with any power of α is possible. The roots of a generator polynomial, $g(X)$, must also be the roots of the codeword generated by $g(X)$, because a valid codeword is of the following form:

$$c(X) = q(X)g(X) \quad (3.4.4)$$

where $q(X)$ is a message-dependent polynomial. Therefore, an arbitrary codeword, when evaluated at any root of $g(X)$, must yield zero, or in other words

$$g(\alpha^i) = c_{\text{valid}}(\alpha^i) = 0, \text{ where } i = 1, 2, \dots, 2t \quad (3.4.5)$$

3.4.2. Implementation of Encoding

A general circuit for parity calculation in encoder for RS codes is shown in Figure 3.4.3.

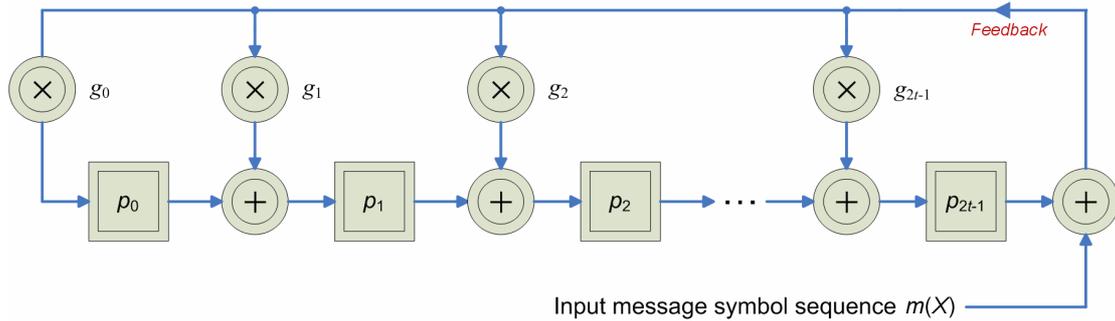
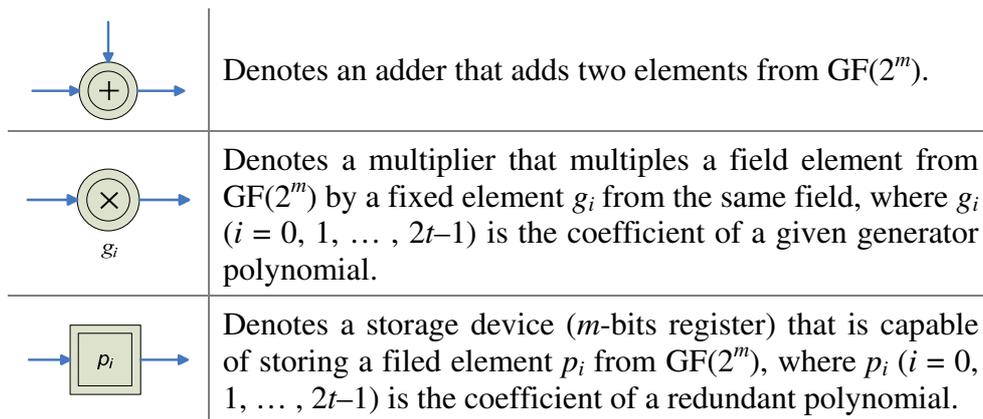


Figure 3.4.3: LFSR encoder for a RS code.

It's a linear feedback shift register (LFSR) circuit, or sometimes called, *division circuit*, where



After the information is completely shifted into the LFSR input, the contents of the registers form the parity symbols. Notice that the arithmetic operators carry out finite field addition or multiplication on complete symbols. For more information, see [8] or [11].

3.5. Reed-Solomon Decoding

When a received codeword is fed to the RS decoder at the receiver for processing, the decoder first tries to verify whether this codeword appears in the dictionary of valid codewords. If it does not, errors must have occurred during transmission over a communication channel. This part of the decoder processing is called *error detection*. If errors are detected, the decoder attempts a reconstruction. This is called *error correction*.

Figure 3.5.1 shows the block diagram of a decoder for RS codes. The decoder consists of digital circuits and processing elements to accomplish the following tasks:

- Compute the *syndromes*;
- Find the coefficients of the *error-location polynomial* $\sigma(X)$.
- Find the *inverses of the roots of* $\sigma(X)$, that is, the locations of the errors;
- Find the *values of the errors*;
- *Correct the received codeword* with the error locations and values found.

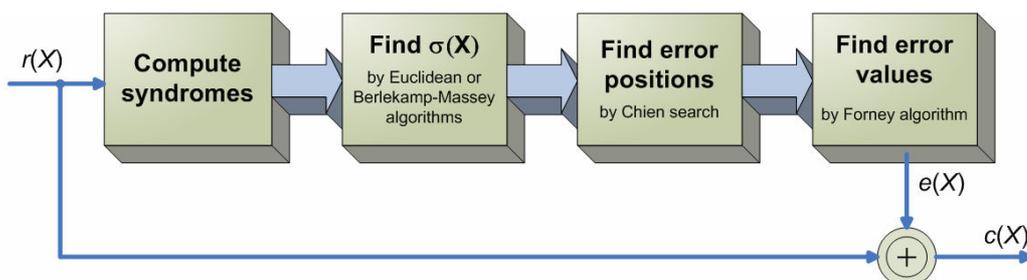


Figure 3.5.1: Architecture of a RS decoder with $GF(2^m)$ arithmetic.

3.5.1. Syndrome Calculation

The syndrome accumulate is the first step in the RS decoding process. This is done to detect if there are any errors in the received codeword.

After encoding a given message, the codeword polynomial

$$c(X) = c_0 + c_1X + \dots + c_{n-1}X^{n-1}$$

is transmitted and affected by noise, and converted into a received polynomial $r(X)$:

$$r(X) = r_0 + r_1X + \dots + r_{n-1}X^{n-1}$$

which is related to the *error polynomial* $e(X)$ and the codeword polynomial $c(X)$ as follows:

$$r(X) = c(X) + e(X) \quad (3.5.1)$$

where the error pattern $e(X)$ added by the channel is expressed as

$$e(X) = r(X) - c(X) = e_0 + e_1X + \dots + e_{n-1}X^{n-1}$$

where $e_i = r_i - c_i$ is a symbol from $GF(2^m)$.

From Expression (3.4.4) it can be seen that every valid codeword polynomial $c(X)$ is a multiple of the generator polynomial $g(X)$. Therefore, the roots of $g(X)$ must also be the roots of $c(X)$. Since $r(X) = c(X) + e(X)$, then $r(X)$ evaluated at each of the roots of $g(X)$ should yield zero only when it is a valid codeword. Any errors will result in one or more of the computations yielding a non-zero result. So the computation of a *syndrome symbol* can be described as follows:

$$S_i = r(\alpha^i), \quad i = 1, 2, \dots, 2t \quad (3.5.2)$$

where $\alpha, \alpha^2, \alpha^3, \dots, \alpha^{2t}$ are the roots of $g(X)$. If $r(X)$ were a valid codeword, it would cause each syndrome symbol S_i to equal 0, or, if one or more syndromes are non-zero, errors have been detected.

The syndrome computation can be accomplished with a division circuit [8], shown in Figure 3.5.2.

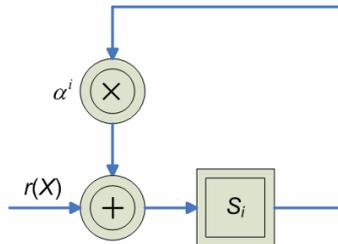


Figure 3.5.2: Syndrome computation circuit for RS codes over $GF(2^m)$.

3.5.1.1. Error Locations and Error Values

Let us assume that the error polynomial $e(X)$ contains $v \leq t$ non-zero elements, which means that during transmission v errors occurred, placed at positions X^{j_1} , X^{j_2} , ..., X^{j_v} , where $0 \leq j_1 < j_2 < \dots < j_v \leq (n-1)$. Then

$$e(X) = e_{j_1} X^{j_1} + e_{j_2} X^{j_2} + \dots + e_{j_v} X^{j_v}$$

The indices 1, 2, ..., v refer to the first, second, ..., v^{th} errors, and the index j refers to the error location. Hence, to correct the corrupted codeword, we need to determine $e(X)$, or in other words, we need to determine the error locations X^{j_i} and the error values e_{j_i} .

Let us now define the error-location number as

$$\beta_l = \alpha^{j_l}, \text{ where } l = 1, 2, 3, \dots, v$$

Next, the $2t$ syndrome symbols are obtained by substituting α^i into the received polynomial for $i = 1, 2, \dots, 2t$:

$$\begin{aligned} S_1 = r(\alpha) &= e_{j_1} \beta_1 + e_{j_2} \beta_2 + \dots + e_{j_v} \beta_v \\ S_2 = r(\alpha^2) &= e_{j_1} \beta_1^2 + e_{j_2} \beta_2^2 + \dots + e_{j_v} \beta_v^2 \\ &\vdots \\ S_{2t} = r(\alpha^{2t}) &= e_{j_1} \beta_1^{2t} + e_{j_2} \beta_2^{2t} + \dots + e_{j_v} \beta_v^{2t} \end{aligned} \quad (3.5.3)$$

In Expression (3.5.3), there are $2t$ unknowns (t error values and t locations), and $2t$ simultaneous equations. However, these $2t$ simultaneous equations cannot be solved in the usual way because they are non-linear (as some of the unknowns have exponents). Any technique that solves this system of equations is known as a *Reed-Solomon decoding algorithm*, since solving the system of equations (3.5.3) constitutes the most computationally intensive operation in decoding RS codes. According to the limitations of the project (Section 1.4), we consider only the two commonly used RS decoding algorithms:

1. *Berlekamp-Massey algorithm*. This is a computationally efficient method in terms of the number of operations in $\text{GF}(2^m)$;
2. *Euclidean algorithm*. This one is less efficient, but tends to be more widely used in practice because it is easier to implement [10].

Notice that all the elements involved in the computation of (3.5.3) belong to a Galois field, and so the operations of addition and multiplication are also done over $\text{GF}(2^m)$.

3.5.2. Berlekamp-Massey Algorithm

If an error has been received, first of all it is necessary to learn the location of the error(s). The syndrome equations in (3.5.3) can be translated into a series of linear equations by defining the *error-location polynomial*:

$$\begin{aligned}\sigma(X) &= (1 + \beta_1 X)(1 + \beta_2 X) \dots (1 + \beta_v X) \\ &= 1 + \sigma_1 X + \sigma_2 X^2 + \dots + \sigma_v X^v\end{aligned}\quad (3.5.4)$$

The roots of $\sigma(X)$ are $1/\beta_1, 1/\beta_2, \dots, 1/\beta_v$. The reciprocal of the roots of $\sigma(X)$ are the error-location numbers of the error pattern $e(X)$. Note that $\sigma(X)$ is an unknown polynomial whose coefficients must be determined. The Berlekamp-Massey algorithm basically consists of finding the coefficients of the error-location polynomial $\sigma(X)$.

The coefficients of $\sigma(X)$ and the error-location numbers are related by the following equations:

$$\begin{aligned}\sigma_0 &= 1 \\ \sigma_1 &= \beta_1 + \beta_2 + \dots + \beta_v \\ \sigma_2 &= \beta_1\beta_2 + \beta_2\beta_3 + \dots + \beta_{v-1}\beta_v \\ &\vdots \\ \sigma_v &= \beta_1\beta_2 \dots \beta_v\end{aligned}\quad (3.5.5)$$

This set of equations is known as the *elementary symmetric functions* and is related to the system of equations (3.5.3), where error values are assumed to have unit magnitude to keep the following equations simple to understand. This relation is

$$\begin{aligned}S_1 + \sigma_1 &= 0 \\ S_2 + \sigma_1 S_1 &= 0 \\ S_3 + \sigma_1 S_2 + \sigma_2 S_1 + \sigma_3 &= 0 \\ &\vdots \\ S_{v+1} + \sigma_1 S_v + \dots + \sigma_{v-1} S_2 + \sigma_v S_1 &= 0\end{aligned}\quad (3.5.6)$$

where S_i are the syndrome symbols. These equations are called *Newton's identities*. The Berlekamp-Massey algorithm is an iterative way to find a *minimum-degree polynomial* that satisfies the Newton's identities. The minimum-degree polynomial $\phi_i(X)$ of an element $\alpha^i \in \text{GF}(2^m)$ is the smallest degree polynomial that has α^i as a root. This is the same as to say that $\phi_i(\alpha^i) = 0$.

The algorithm proceeds as follows: The first step of iteration is to determine a minimum-degree polynomial $\sigma_1(X)$ whose coefficients satisfy the first Newton's identity described in (3.5.6). Then the second Newton's identity is tested, that is, if the polynomial $\sigma_1(X)$ also satisfies the second Newton's identity in (3.5.6), then

$\sigma_2(X) = \sigma_1(X)$. Otherwise, the decoding procedure adds a correction term to $\sigma_1(X)$ in order to form the polynomial $\sigma_2(X)$, which is able to satisfy the first two Newton's identities. This procedure is subsequently applied to find $\sigma_3(X)$ and the following polynomials, until determination of the polynomial $\sigma_{2t}(X)$ is complete. Once the algorithm reaches this step, the polynomial $\sigma_{2t}(X)$ is adopted as the error-location polynomial $\sigma(X)$, that is, $\sigma(X) = \sigma_{2t}(X)$, since this last polynomial satisfies the whole set of Newton's identities of (3.5.6). After the determination of the error-location polynomial, the roots of this polynomial are calculated by applying the *Chien search*.

If the degree of $\sigma(X)$ obtained by the Berlekamp-Massey algorithm exceeds t , this indicates that more than t errors have occurred and the codeword is therefore not correctable.

The summary of the iterative steps of the Berlekamp-Massey algorithm can be found in [8] or [10].

3.5.3. Euclidean Algorithm

We have seen that the Berlekamp-Massey algorithm can be used to construct the error-location polynomial. In this section, we show that the Euclidean algorithm can also be used to construct error-location polynomials.

The Euclidean algorithm is a recursive technique to find the *Greatest Common Divisor (GCD)* between two polynomials (or integers). A common divisor $g > 0$ such that every common divisor of a and b divides g is called the greatest common divisor (GCD) and is denoted by (a, b) . For polynomials, if either $a(X)$ or $b(X)$ is not zero, the common divisor $g(X)$ such that every common divisor of $a(X)$ and $b(X)$ divides $g(X)$ is referred to as GCD of $a(X)$ and $b(X)$ and is denoted by $(a(X), b(X))$.

The main idea of the Euclidean algorithm is that it works by simple repeated division: Starting with two numbers, a and b , divide a by b to obtain a remainder. Then divide b by the remainder, to obtain a new remainder. Proceed in this manner, dividing the last divisor by the most recent remainder, until the remainder is 0. Then the last non-zero remainder is the greatest common divisor (a, b) .

The Euclidean algorithm for decoding RS codes is established with the help of the following theorem: *If $g = (a, b)$ then there exist integers s and t such that*

$$g = (a, b) = as + bt$$

For polynomials, if $g(X) = (a(X), b(X))$, then there are polynomials $s(X)$ and $t(X)$ such that

$$g(X) = a(X)s(X) + b(X)t(X) \quad (3.5.7)$$

So the Euclidean algorithm in RS decoding computes $g(X) = (a(X), b(X))$ and also $s(X)$ and $t(X)$, and is sometimes called the *extended Euclidean algorithm*.

Coming back to the decoding of RS codes, we are able to re-write (3.5.7) in the following form:

$$\Lambda(X) = X^{2t}\mu(X) + S(X)\sigma(X) \quad (3.5.8)$$

where $\Lambda(X)$ is called the *error-evaluation polynomial* and plays the role of $g(X)$ in (3.5.7), X^{2t} plays the role of $a(X)$, $\mu(X)$ is some polynomial and plays the role of $s(X)$, the syndrome polynomial $S(X)$ plays the role of $b(X)$, and the error-location polynomial $\sigma(X)$, which is what we want to know, plays the role of $t(X)$.

The error-evaluation polynomial $\Lambda(X)$ is expressed as

$$\Lambda(X) = (\sigma(X)S(X)) \bmod X^{2t} \quad (3.5.9)$$

The syndrome polynomial $S(X)$ is described as

$$S(X) = S_1 + S_2X + S_3X^2 + \dots + S_{2t}X^{2t-1} \quad (3.5.10)$$

where $S_1, S_2, S_3, \dots, S_{2t}$ are the syndrome symbols calculated as (3.5.2).

Now keeping in mind that (3.5.8) in our case is equivalent to (3.5.7), let's explain the steps of the extended Euclidean algorithm with the notation of (3.5.7). The following explanation is extracted from [7].

The algorithm involves repeated division of polynomials until a remainder of degree $< t$ is found. In order to perform this division, the *long division* technique is used.

The first step is to divide $a(X)$ by $b(X)$ to find the quotient $q_1(X)$ and remainder $r_1(X)$, such that:

$$a(X) = q_1(X)b(X) + r_1(X) \quad (3.5.11)$$

If the degree of $r_1(X)$ is less than t , then we have reached our solution with $s(X) = 1$, $t(X) = q_1(X)$ and $g(X) = r_1(X)$. Otherwise, set $t_1(X) = q_1(X)$ and proceed to the next stage.

The second step is to divide $b(X)$ by $r_1(X)$ giving

$$b(X) = q_2(X)r_1(X) + r_2(X) \quad (3.5.12)$$

Note that the degree of $r_2(X)$ must be less than that of $r_1(X)$ so that this process is reducing the degree of the remainder. If we eliminate $r_1(X)$ from Equations (3.5.11) and (3.5.12) we obtain

$$q_2(X)a(X) = [q_2(X)t_1(X) + 1]b(X) + r_2(X) \quad (3.5.13)$$

Set $t_2(X) = q_2(X)t_1(X) + 1$. If the degree of $r_2(X)$ is less than t , then $t(X) = t_2(X)$; otherwise, continue to the next step.

The third step continues in similar way, dividing $r_1(X)$ by $r_2(X)$:

$$r_1(X) = q_3(X)r_2(X) + r_3(X)$$

Again the degree of the remainder is decreasing. Using Equations (3.5.12) and (3.5.13) to eliminate $r_1(X)$ and $r_2(X)$, gives

$$[1 + q_2(X)q_3(X)]a(X) = [t_1(X) + q_3(X)t_2(X)]b(X) + r_3(X)$$

If the degree of $r_3(X)$ is less than t , then $t(X) = t_3(X) = q_3(X)t_2(X) + t_1(X)$.

The method continues in this way until a remainder of degree less than t is found, at each stage $i = (1, 2, \dots)$ setting

$$t_i(X) = q_i(X)t_{i-1}(X) + t_{i-2}(X) \quad \text{with } \{t_0(X) = 1, t_{-1}(X) = 0\} \quad (3.5.14)$$

So the last calculated $t_i(X)$ corresponds to the desired error-location polynomial $\sigma(X)$ in (3.5.8). Roots of the polynomial $\sigma(X)$ can be obtained by using the Chien search, as described in the following section.

3.5.4. Chien Search

The next step in the RS decoding process is to find the roots of the determined error-location polynomial $\sigma(X)$, which is a polynomial whose roots are constructed to be the reciprocal of the locations where the errors occurred. There is no closed form solution for solving for the roots of $\sigma(X)$. Since the root obviously has to be one of the elements of the field $\text{GF}(2^m)$, an exhaustive search by substituting each of the finite field elements in the error-location polynomial $\sigma(X)$ and checking for the condition $\sigma(\alpha^i) = 0$ is the only way out. The Chien search is an effective algorithm to do this exhaustive search in an efficient manner.

Suppose, for example, that $v = 3$ and the error-location polynomial is

$$\sigma(X) = 1 + \sigma_1 X + \sigma_2 X^2 + \sigma_3 X^3$$

We evaluate $\sigma(X)$ at each non-zero element in $\text{GF}(2^m)$ in succession:

$$X = 1, \quad X = \alpha, \quad X = \alpha^2, \quad \dots, \quad X = \alpha^{2^m-2}$$

This gives us the following:

$$\begin{aligned} \sigma(1) &= 1 + \sigma_1(1) + \sigma_2(1)^2 + \sigma_3(1)^3 \\ \sigma(\alpha) &= 1 + \sigma_1(\alpha) + \sigma_2(\alpha)^2 + \sigma_3(\alpha)^3 \\ \sigma(\alpha^2) &= 1 + \sigma_1(\alpha^2) + \sigma_2(\alpha^2)^2 + \sigma_3(\alpha^2)^3 \\ &\vdots \\ \sigma(\alpha^{2^m-2}) &= 1 + \sigma_1(\alpha^{2^m-2}) + \sigma_2(\alpha^{2^m-2})^2 + \sigma_3(\alpha^{2^m-2})^3 \end{aligned}$$

If in the above performed substitutions we obtain $\sigma(\alpha^i) = 0$, then the exponent of the inverse of the root α^i is equal to the error-location index i .

3.5.5. Forney Algorithm

Having found the error-location polynomial and its roots, there is still one more step in the RS decoding: we have to find the error values. In general, this is done using the *Forney algorithm*, where the error value at location $\beta_l = \alpha^{j_l}$ ($1 \leq l \leq \nu$) for a RS code is computed by

$$e_{j_l} = \frac{\Lambda(\beta_l^{-1})}{\sigma'(\beta_l^{-1})} \quad (3.5.15)$$

where $\Lambda(X)$ is the error-evaluation polynomial, described as (3.5.9), and $\sigma'(X)$ is the formal derivative of the error-location polynomial $\sigma(X)$ with respect to X . It turns out that the derivative of any polynomial in the finite field is simple to compute as odd powers can be zeroed out and even powers shifted down by one. This enables easy computation of the error values.

So having found the error locations and error values, we finally can form the error polynomial $e(X)$ and correct the received polynomial $r(X)$ just by adding (with XOR operation) these two polynomials together, as shown in Figure 3.5.1.

3.6. Summary

This chapter summarized the essence of RS codes and provided background information on finite fields known as Galois fields (the codes are based on the use of Galois field arithmetic). Furthermore, this chapter introduced the general concept of Galois field arithmetic implementation. Finally, the principles of RS encoding and decoding were explained. With RS decoding, the two commonly used RS decoding algorithms were presented: Berlekamp-Massey and Euclidean decoding algorithms.

Chapter 4

PERFORMANCE EVALUATION OF REED-SOLOMON CODES

Before a real-time implementation is initiated, it is very helpful to perform a *simulation* of the given system: when the system is simulated and the practical performance is obtained, it is necessary to compare the obtained performance with the theoretical one to ensure that the system at hand works correctly. However, in order to simulate a particular system, first of all a *model* of that system should be developed.

This chapter introduces the conceptual modeling and simulation of Reed-Solomon (RS) codes used in ADSL. Moreover, the simulation results are compared with the theoretical performance of RS codes. Finally, a decision on which RS code to use in its further implementation on the target architectures is made.

4.1. Theoretical Performance of Reed-Solomon Codes

According to the standard decoding algorithm of RS code, if the number of symbol errors in the received codeword is not larger than t (error correcting capability), the decoder can correct all of them. When the number of symbol errors is larger than t , the decoder either provides a *mis-decoding* result or declares *decoding failure* and passes the (uncorrectable) codeword unchanged. The concept of mis-decoding is described in Section 4.1.4.

4.1.1. Code Rate

The *code rate* of a code is given by:

$$\text{code rate} = r = \frac{k}{n} \quad (4.1.1)$$

where k is the number of information (message) symbols per codeword, and n is total number (information + redundancy) of code symbols per codeword. This definition holds for all codes whether RS codes or not.

Lower rate codes, characterized by small values of r , can generally correct more channel errors than higher rate codes and are thus more energy efficient. However, higher rate codes are more bandwidth efficient than lower rate codes, because the amount of overhead (in the form of parity symbols) is lower. Thus there is a trade-off between energy efficiency and bandwidth efficiency.

4.1.2. Reed-Solomon Performance as a Function of Code Size

For a code to successfully manage the effects of noise, the noise duration has to represent a relatively small percentage of the codeword. To ensure that this happens most of the time, the received noise should be averaged over a long period of time. Hence, error-correcting codes become more efficient (performance improves) as the codeword size increases (keeping the constant code rate). This is seen by the family of curves in Figure 4.1.1, where the code rate is held at a constant ($k/n = 0.92$), while its codeword size increases from $n = 51$ symbols to $n = 255$ symbols (with $m = 8$ bits per symbol). Here, the codes with $n < 255$ are the shortened codes. On the other hand, as the codeword size increases (keeping the same code rate), the implementation of RS codes grows in complexity.

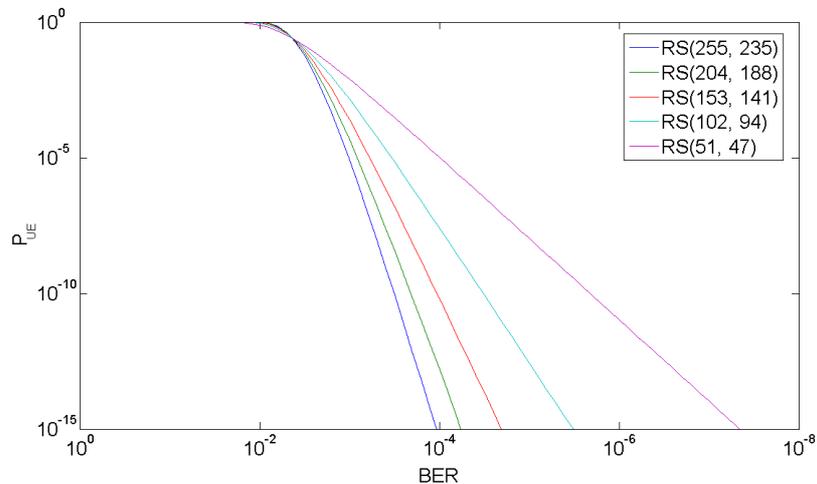


Figure 4.1.1: Performance curves for different RS codes of rate 0.92. The shown RS codes are from the typical range, which can be found in the literature.

Figure 4.1.1 presents the RS code performance in terms of P_{UE} and channel bit error rate (BER), where P_{UE} is the probability of an uncorrectable error, i.e., is the ratio of the number of uncorrectable codewords to the total number of received codewords, in the limiting case where the number of codewords received becomes large. Assuming that the symbol errors are independent and that no erasure information is available, the probability of an uncorrectable error for RS codes can be expressed as: [12]

$$P_{UE} = 1 - \sum_{i=0}^t C_n^i (P_{SE})^i (1 - P_{SE})^{n-i} \quad (4.1.2)$$

where n is the number of symbols per codeword, C_n^i is the binomial coefficient, and P_{SE} is the channel symbol error rate. The binomial coefficient is evaluated as

$$C_n^i = \frac{n!}{i!(n-i)!}, \text{ where } n! = \prod_{i=1}^n i$$

P_{SE} is the probability that the channel will change a symbol during the transmission of the message. Under the assumption of purely random bit errors, we can write: [12]

$$P_{SE} = 1 - (1 - P_B)^m$$

where m is the number of bits per symbol, and P_B is the channel BER.

In general, RS codes perform better as the bit error pattern becomes less random. The formulas presented in this section generally predict larger error probabilities than will be encountered with correlated or burst-type error patterns.

4.1.2.1. Selection of Reed-Solomon Codeword Size

Now taking into account that in our case the structure of RS codes can be as complex as required to achieve the highest available bit-error performance (according to the system requirements described in Section 1.5), and taking into account that the size of codewords should be fixed (according to the system constraints described in Section 1.4), we select the maximum RS codeword size supported by ADSL, that is, $n = 255$ symbols per codeword. As we see in Figure 4.1.1, the performance curve for the RS(255, 235) code, where $n = 255$, provides the best result in terms of BER and uncorrectable error as compared with other curves. So, a class of RS(255, k) codes is selected for its further analysis, simulation and implementation.

4.1.3. Reed-Solomon Performance as a Function of Redundancy

As we know from Chapter 2, on the transmitter side a forward error correcting (FEC) encoder adds redundancy to the data in the form of parity information. Then at the receiver a FEC decoder is able to exploit the redundancy in such a way that a reasonable number of channel errors can be corrected. In general, an RS decoder can detect and correct up to t incorrect symbols in a received codeword if there are $2t$ redundant symbols in the encoded message. Hence, the higher redundancy is (lower code rate), the more erroneous symbols can be corrected in a codeword, resulting in better bit-error performance. However, as the redundancy of a RS code increases, its implementation grows in complexity. Also, the bandwidth expansion must grow for any real-time communications application.

The curves in Figure 4.1.2 depict the probability of uncorrectable error P_{UE} for a codeword size n fixed to 255, and t varying from 1 to 8 (what is supported by ADSL), as a function of the channel BER.

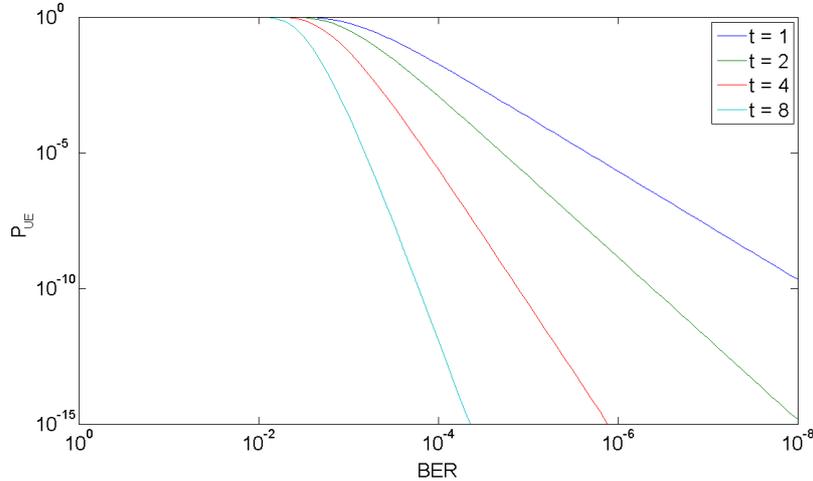


Figure 4.1.2: *RS(255, 255-2t) code performance as a function of redundancy.*

Notice that for BER below 10^{-2} , the curve with $t = 8$ (Figure 4.1.2) exhibits a very steep slope. This is characteristic for good codes. This steep slope is preferred for data communications, since large improvements in output P_{UE} are possible for small improvements in input BER. [12]

4.1.4. Mis-decoding

“Mis-decoding” is the name given to a wrong error detection and correction (EDAC) operation. In mis-decoding the received codeword contains a combination of errors such that the decoder misinterprets the situation and performs a mistaken correction (a “mis-decode”), which yields a totally wrong message codeword at the receiver.

From Section 3.2 we know that the simultaneous RS error-correction and erasure-correction capability can be expressed as

$$2n_{errors} + n_{erasures} \leq 2t \quad (4.1.3)$$

where n_{errors} is the number of errors with an unknown location, and $n_{erasures}$ is the number of errors with known locations (erasures). So the RS algorithm guarantees to correct a codeword, if the condition (4.1.3) is true. But if there are so many errors that this condition is not met, one of two situations occurs:

1. The error is properly detected by the decoder and becomes a detected error (decoding failure is declared), or
2. The erroneous message appears to the decoder as a correctable error and the error is corrected to the wrong codeword and becomes a decoding error (this is a mis-decoding).

In order to handle the second situation, an additional error detection operation is required, such as CRC, which was described in Section 2.2.3.1.

For a reasonably high value of t (≥ 8) and $n \geq 5t$, the probability of mis-decoding is much smaller than that of decoding failure, and hence, can be ignored [13]. In such case, due to the excellent error detection capability of a RS code, no additional operation, such as CRC, is required for error detection.

4.2. Simulation of Reed-Solomon Codes

Before the practical implementation on the target architecture is initiated, it is necessary to perform a simulation of the given system to verify its functionality. In order to simulate a particular system, first of all a model of that system should be developed.

4.2.1. FEC Model

A system model tries to mimic some properties of a system. In order to model the static and dynamic properties of a system in a structured way, we supplement the mathematical framework by the notion of *system models*. A system model characterizes an abstract view of the systems under development. A system model both describes the static structure of its components and their behavior over time.

Our model corresponds to the FEC communication scheme, which is based on a RS error-control code, and illustrated in Figure 4.2.1.

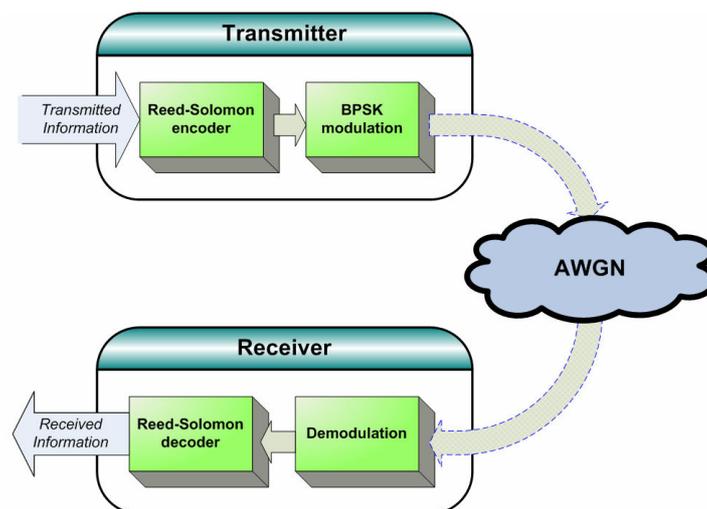


Figure 4.2.1: FEC model based on a RS code.

The behavior of each block of the model, shown in Figure 4.2.1, is described in C programming language. The corresponding source code can be found in the enclosed CD of the report. A short description of each block is presented below:

- **Reed-Solomon encoder/decoder (RS codec):**

The RS codec has three parameters, which can be modified:

1. m = the number of bits per symbol;
2. n = the codeword size in symbols;
3. red = the number of redundant symbols.

The Berlekamp-Massey algorithm is a popular choice to simulate RS decoders in software, because it is very efficient in terms of the number of $GF(2^m)$ operations. It must be noted that the Berlekamp-Massey algorithm is more efficient (in terms of operations) than the Euclidean algorithm [10]. Therefore, the Berlekamp-Massey algorithm is used in our model (Figure 4.2.1) to be simulated.

- **BPSK modulation/demodulation:**

As we know from Section 3.2.1, when a RS decoder corrects a symbol, it replaces the incorrect symbol with the correct one, whether the error was caused by one bit being corrupted or all symbol bits being corrupted. Thus, in case of simulation, it is not necessary to transmit all symbol bits over the channel. It is enough to transmit only one symbol bit and disturb it by the effects of the channel, since the simulation results will be the same whether we corrupt only one or all symbol bits. In such case, we obtain binary transmission over the channel.

With binary transmission, it is convenient to use a binary modulation. In our model (Figure 4.2.1), a *Binary Phase Shift Keying (BPSK) modulation* [14] is used, as it is easy to implement: in BPSK modulation, each data bit is transformed into a separate channel symbol (real amplitude): if the binary data value is 1, the channel symbol is -1 , and if the binary data value is 0, the channel symbol is $+1$.

As mentioned in Section 2.1, the frequency at which bit errors occur at the output of the FEC decoder is a measure of the demodulator-decoder performance. This is valid when *multilevel modulation* [14] is used. With binary modulation, the frequency at which bit errors occur is a measure only of the decoder performance. Thus, using BPSK modulation, the simulation results in our case will depend only on the RS decoder performance.

- **AWGN channel:**

Coding performance curves are regularly shown for the AWGN channel. There are two reasons why this is so. First, burst-error mechanisms are often badly understood and there may be no generally accepted models that fit the real behaviour. The other reason is that most codes in use are primarily designed for random error channels (the only important codes where this is not the case are RS codes). For that reason, the AWGN channel is accepted as the basic model for a digital communication channel and therefore used as a standard channel model in our case.

With the given model to be simulated (Figure 4.2.1), the data is transmitted over the AWGN channel, shown in Figure 4.2.2.

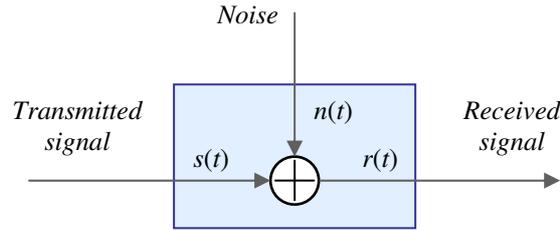


Figure 4.2.2: AWGN channel.

In the AWGN channel, the transmitted signal $s(t)$ gets disturbed by a simple additive white Gaussian noise (random noise) process $n(t)$, and the received signal $r(t)$ is given by

$$r(t) = s(t) + n(t)$$

As RS codes are particularly designed for correcting burst errors (but not random errors), the worst bit-error performance of RS codes can be evaluated when using the AWGN channel.

4.2.2. FEC Model Simulation

After a model is developed, it should be verified to ensure that its functionality is proper. It can be done by using a simulation mechanism. The main goal of simulation is to further the understanding of model algorithms and data structures inductively, based on observations of an algorithm in operation.

4.2.2.1. Simulation Results

In order to simulate the FEC model, shown in Figure 4.2.1, the RS codec parameters are set to:

- $m = 8$;
- $n = 255$;
- red (the number of redundant symbols) = 2, 4, 6, 8, 10, 12, 14, 16 (what is supported by ADSL), and 32.

The simulation results are depicted in Figure 4.2.3. From the simulation results, we observe that the performance of RS codes in terms of BER improves with the increase in redundancy. According to Section 4.1.3, this confirms the correctness of the model functionality. Furthermore, in Figure 4.2.3, we notice that the performance curves with $t = 1 \dots 8$ reach particular E_b/N_0 values and then no longer have been improved in terms of BER. This indicates that the curves with $t = 1 \dots 8$ are not able to fully cope with the present noise level, that is, the error correcting capability of the corresponding RS codes is not high enough to correct all the errors introduced by the channel. The only way out is to greatly increase the transmitted signal power to

reduce the non-desired effects of the channel. For example, the performance curve with $t = 16$ (Figure 4.2.3) is able to totally remove all the effects of the channel in received codewords starting at E_b/N_0 of 13.5 dB.

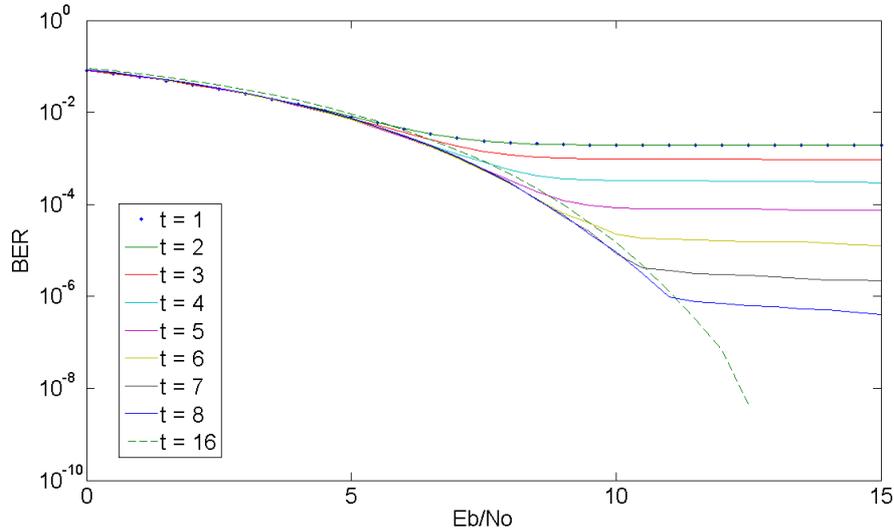


Figure 4.2.3: BER vs. SNR per bit (E_b/N_0) performance of RS(255, 255-red) for BPSK modulation, where $red = 2t$.

4.2.3. Selection of Reed-Solomon Redundancy

Now taking into account that in our case the structure of RS codes can be as complex as required to achieve the highest available bit-error performance (according to the system requirements described in Section 1.5), and taking into account that the number of redundant symbols should be fixed (according to the system constraints described in Section 1.4), we select the maximum RS redundancy supported by ADSL, that is, $2t = 16$ redundant symbols per codeword. As we see in Figures 4.1.2 and 4.2.3, the performance curves for the RS(255, 239) code, where $2t = 16$, provide the best bit-error performance as compared with other curves. So, the RS(255, 239) code is selected for its further analysis and implementation on the target architectures.

4.2.3.1. Advantages and Drawbacks of RS(255, 239) in ADSL

The RS(255, 239) code is the most complex code in comparison with the other supported by ADSL RS codes, as the codeword size and redundancy of RS(255, 239) are the maximum available in ADSL. This certainly leads to high implementation complexity. However, we can take several advantages of RS(255, 239) for ADSL:

- The ADSL system is standardized to work at BER of 10^{-7} (utilizing both Trellis coding and RS FEC) [1]. As we see in Figure 4.2.3, the performance curve of RS(255, 239) almost reaches this BER in case of random noise (is the worst case for RS codes). It means that if E_b/N_0 , at which RS(255, 239) provides the best BER, is satisfied, Trellis coding in ADSL can be eliminated;

- With the RS(255, 239) code, the probability of mis-decoding is much smaller than that of decoding failure, and hence, can be ignored (see Section 4.1.4). In such case, the CRC operation in an ADSL modem can be eliminated.

4.3. Summary

This chapter introduced the conceptual modeling and simulation of the RS codes used in ADSL. Moreover, the simulation results were compared with the theoretical performance of RS codes to ensure that the system works correctly. Besides, according to the system constraints described in Section 1.4, and according to the system requirements described in Section 1.5, the RS(255, 239) code was selected over another from the simulation results for its further analysis and implementation on the target architectures. Finally, the advantages and drawbacks of RS(255, 239) for ADSL were presented.

Chapter 5

ALGORITHM CHARACTERIZATION

As mentioned in Section 1.3, the main idea of *algorithm characterization* is to extract relevant information from the specification of an application (i.e., algorithm) to guide the designer towards an efficient algorithm-architecture matching. For this purpose, different metrics can be efficiently used to rapidly stress the proper architecture style for the given algorithm. Furthermore, these metrics can be properly combined in order to build a *global metric* able to suggest the most suitable type of architecture for the execution of an algorithm.

To obtain a global metric, several subtasks should be performed: 1) an analysis of the available architectures to determine their relevant features; 2) the definition of a set of patterns able to identify subsets of the algorithm specification that could exploit the identified architectural features, and 3) the definition of a set of metrics able to provide meaningful indications useful to make design choices. Each of these subtasks is presented below in the current chapter.

5.1. Architectural Features

This section describes the analysis performed to detect the most relevant exploitable architectural features of the given *processing elements* (i.e., DSP and FPGA).

5.1.1. DSP Architectural Features

Digital Signal Processor (DSP) is a microprocessor specifically designed to handle digital signal processing (i.e., stream-based processing) tasks, such as filtering or correlation. Current DSPs, involving advanced addressing, efficient interfaces, and powerful functional units, such as fast multipliers and barrel shifters, provide superior performances in limited application spaces. Many DSP processors today employ an internal *Harvard Architecture* (the same for the given DSP, see Appendix B). The Harvard architecture uses different memories for their instructions and data, requiring dedicated buses for each of them. Hence, the architectural features included in the given DSP allow concurrent fetching of instructions and operands, concurrent

execution of arithmetic operations (e.g., sums and multiplications), fast management of loops, and fast access to sequential memory location (e.g., array). Further info can be found in [15], [16] and Appendix B.

5.1.2. FPGA Architectural Features

A Field-Programmable Gate Array (FPGA) is a large-scale integrated circuit that can be programmed after it is manufactured rather than being limited to a predetermined, unchangeable hardware function. The term “field-programmable” refers to the ability to change the operation of the device “in the field”, while “gate array” is a somewhat dated reference to the basic internal architecture that makes this after-the-fact reprogramming possible. An FPGA device, with its inherently scalable *parallelism*, can be used to implement just about any hardware design. The ability to manipulate FPGA logic at the gate level allows designers to create a custom processor that can efficiently implement exactly the function the application requires, simultaneously performing N application subfunctions in parallel. As N increases, the advantage of FPGAs becomes even more significant. Further info can be found in Appendix A.

Moreover, FPGA devices are more suitable to perform *bit-manipulation operations* (Boolean operators, shifting, etc.) than DSPs. [15]

5.2. Performance Metrics

Performance metrics are the mean of evaluating a given design specification to test its particular properties. Considering the architectural features previously described, i.e.:

- in case of DSP: *circular addressing*, *Multiply and Accumulate (MAC) operations*, and *Harvard architecture*, and
- in case of FPGA: *inherent parallelism* and *bit-manipulation operations*,

it is possible to define a set of patterns able to identify subsets of the specification that match some of these features, and a set of metrics that quantify such matching. Finally, these metrics can be properly combined in order to build a global metric, called *affinity* [15], able to suggest the most suitable processing element for the execution of a given algorithm.

5.2.1. Data Oriented Metric

The goal of this metric is to take into account the type of data involved in the execution of an algorithm (a more clear idea of this metric will be presented later):

- **Data Ratio ($DR_{m,t}$)**. “For each method (or function) m and for each allowed type t (e.g., `int`, `float`, etc.), $DR_{m,t}$ is defined as the fraction of declarations of type t with respect to the total number of declarations made in m ” [15].

5.2.2. DSP Oriented Metrics

The purpose of DSP oriented metrics is to identify functionalities suitable to be executed by a DSP device by considering those issues that exploit the most relevant architectural features of such processing element: circular addressing, MAC operations, and Harvard architecture.

5.2.2.1. Circular Addressing

Circular addressing is used to create a circular buffer. The buffer is created in DSP memory and is very useful in such DSP algorithms, as filtering or correlation, where data needs to be updated. This addressing mode (i.e., circular) is used in conjunction with a circular buffer to update samples by shifting data without the overhead created by shifting data directly. As a pointer reaches the end of a circular buffer that contains the last element (sample) in the buffer and is then incremented, the pointer is automatically wrapped around or points to the beginning of the buffer that contains the first element.

So the use of a circular buffer in an algorithm can be identified (more or less explicitly) by portions of the source code (written in C/C++ language) that try to shift array (i.e., buffer) elements of one or more positions:

- **Strong Circularity Degree (SCD_m)**. “For each method m , SCD_m is the ratio between the number of source lines that contain expressions of the form $v[i] = v[i \pm K]$ and the total number of lines, where v is an array (or a row/column of a matrix), and K is a constant” [15];
- **Weak Circularity Degree (WCD_m)**. “For each method m , WCD_m is the ratio between the number of source lines that contain expressions of the form $v[K] = f(v[i])$ or $q = f(v[i])$ and the total number of lines, where v is an array (or a row/column of a matrix), K is a constant, and $f(v[i])$ is an expression involving $v[i]$ ” [15].

5.2.2.2. MAC Operations

Now consider the C code fragment (Code 5.2.1) that computes the *inner* (or *dot*) *product*, which is useful for many DSP algorithms (e.g., filtering), of two arrays, x and y , each having N elements:

```
result = 0;
for(i=0; i<N; i++)
    result += x[i] * y[i];
```

Code 5.2.1: Sum of products.

A DSP device incorporates several features aimed at optimizing such a loops. These features are usually DSP specific, and with the given DSP, their description can be found in [16]. So the use of a dot product in an algorithm can be identified by

portions of the source code that express a particular mix of operations (i.e., a sum and a multiplication) that DSP can optimize:

- **Strong MAC Degree (SMD_m)**. “For each method m , SMD_m is the ratio between the number of source lines inside a loop that contain expressions of the form $s = s + a_x * a_y$ and the total number of lines” [15];
- **Weak MAC Degree (WMD_m)**. “For each method m , WMD_m is the ratio between the number of source lines that contain, outside a loop, expressions of the form $s = s + a_x * a_y$ and the total number of lines” [15].

5.2.2.3. Harvard Architecture

The Harvard architecture allows a processor to fetch instructions from memory while concurrently reading or writing data to another memory location. Thus, for the concurrent memory access, the goal is to identify subsets of an algorithm able to exploit concurrent memory accesses to data and instructions: [15]

- **Strong Harvard Degree (SHD_m)**. “For each method m , SHD_m is the ratio between the number of source lines that contain, inside a loop, expressions with the structure $v[i] <op> w[i]$ or $q <op> w[i]$ and the total number of lines, where v and w are arrays, and $<op>$ is an operator different from the assignment one (i.e., ‘=’)” [15];
- **Weak Harvard Degree (WHD_m)**. “For each method m , WHD_m is the ratio between the number of source lines that contain, outside a loop, expressions with the $v[i] <op> w[i]$ or $q <op> w[i]$ and the total number of lines, where v and w are arrays, and $<op>$ is an operator different from the assignment” [15].

5.2.3. FPGA Oriented Metrics

The goal of these metrics is to highlight relevant FPGA features: the high degree of inherent parallelism, and fast handling of bit-manipulation operations.

In particular, the inherent parallelism refers to the ability to perform many actions simultaneously, that is, appropriate parts of an algorithm can be executed concurrently in FPGA, which definitely requires more resource usage (e.g., memory elements, logic gates) at a certain time. On the other hand, the more parts of an algorithm are performed in parallel, the higher speed performance of this algorithm can be achieved. Thus, the following two metrics are proposed:

- **Strong Parallelism Degree (SPD_m)**. For each method m , SPD_m is the ratio between the number of source lines inside a loop that can be executed in parallel and the total number of lines;
- **Weak Parallelism Degree (WPD_m)**. For each method m , WPD_m is the ratio between the number of source lines outside a loop that can be executed in parallel and the total number of lines.

For bit-manipulation, the goal is to identify regular functionalities that significantly rely on such operations, as shifting (by a constant), AND, OR, XOR. Therefore, the following metric is defined:

- **Bit Manipulation Rate (BMR_m)**. “For each method m , BMR_m is the ratio between the number of source lines that contain bit-manipulation operations and the total number of lines” [15].

5.2.4. Difference between Strong and Weak Degrees

As can be observed, some of the metrics previously defined are divided into *strong* and *weak* degrees. The difference between these two degrees is that a certain code part (inside a loop) taken into account in calculating the strong degree of a particular architectural feature can benefit more from that feature than a certain code part (outside a loop) taken into account in calculating the corresponding metric of weak degree. Here, with the increase in the number of loop iterations, the benefit from a particular architectural feature could grow as well. For example, as the array length becomes large in Code 5.2.1, the ratio of time spent in such loops to the time spent outside the loops becomes even more significant (see [16]). So, in order to distinguish between code parts that can significantly benefit from a particular architectural feature and code parts that are not able to gain much from the feature, the following steps are proposed.

As was mentioned above, the higher number of loop iterations is, the more an appropriate code part inside that loop can benefit from a particular architectural feature. Now let's divide all loops in a code into *highly computational* and *non-highly computational*. Not making any clear definitions at the moment, let's say that a highly computational loop is a loop that contains a relatively high number of iterations, and may require the most computations in the entire code. So, a non-highly computational loop is a code part different from highly computational loop. Now let's make some corrections in the definitions of the following metrics: SCD_m , SMD_m , SHD_m , SPD_m , WCD_m , WMD_m , WHD_m , WPD_m . The correction within SCD_m , SMD_m , SHD_m , SPD_m is that the specified subsets of the code that can exploit the corresponding architectural features should now be identified inside highly computational loops, and the correction within WCD_m , WMD_m , WHD_m , WPD_m is that appropriate subsets of the code should now be identified outside highly computational loops. Let's denote the modified metrics as $SCD2_m$, $SMD2_m$, $SHD2_m$, $SPD2_m$, $WCD2_m$, $WMD2_m$, $WHD2_m$, $WPD2_m$ (second version of metrics), respectively.

After this correction, we can state that certain code parts (inside highly computational loops) taken into account in calculating $SCD2_m$, $SMD2_m$, $SHD2_m$, $SPD2_m$ will benefit much more from the corresponding architectural features than certain code parts (outside highly computational loops) taken into account in calculating $WCD2_m$, $WMD2_m$, $WHD2_m$, $WPD2_m$. In order to make sure that this is true, and to understand more clearly why this correction was made, let's move to the next section, which includes an analysis of both metric versions.

5.2.5. Selection of Defined Metrics

Once the metrics have been defined, they have to be taken into account in defining the affinity, which towards a certain processing element depends on the degree (strong and weak) of a particular set of metrics. Considering both strong and weak degrees for the affinity, the matching (provided by this affinity) between an algorithm and a certain processing element may become less precise as compared with the case where only the strong degree is considered. The following subsection gives an explanation why this is so.

5.2.5.1. Selection of FPGA Oriented Metrics

Now let's consider and analyze the following C code fragment:

```
void main()
{
    ...

    for(i=0; i<1024; i++) // "First" loop
    {
        a[i] = a[i] + i;    // statement #1
        b[i] = b[i] * 2;    // statement #2
        c[i] = c[i] * 3;    // statement #3
    }

    for(i=0; i<8; i++)    // "Second" loop
    {
        j = i * 64;        // statement #4
        k = i * 128;       // statement #5
        out[i] = a[j]+b[k]+c[k-j]; // statement #6
    }

    out[0] = out[0] * 2;    // statement #7
    out[1] = out[1] * 0.5;  // statement #8
    out[2] = out[2] * 5;    // statement #9
    :
    :
    out[7] = out[7] * 1.3;  // statement #14
}

```

Code 5.2.2: C source code, which is used as the example in the current section.

We can observe that there are two loops ("first" and "second") in the code shown above. Now let us assume that each *code statement* (line of code ending with a semicolon) takes only one clock cycle to be performed in FPGA¹. Hence, the body of the "first" loop takes $1024 * 3 = 3072$ cycles, the body of the "second" loop takes $8 * 3 = 24$ cycles, and the last eight statements (#7, #8, ..., #14) take 8 cycles overall. At this point we can state that the "first" loop is *highly computational*, as it takes almost all cycles of the total number (i.e., $3072 + 24 + 8 = 3104$) of code cycles. Consequently, the "second" loop differs from a highly computational loop, as it takes only 24 cycles out of 3104 code cycles. Therefore, considering the correction made in Section 5.2.4, the "first" loop should be involved in computing the strong parallelism

¹ As with Handel-C language, see Chapter 6.

degree (SPD_{2_m}) metric, and the “second” loop with statements #7...#14 should be involved in computing the weak parallelism degree (WPD_{2_m}) metric. With SPD_m and WPD_m metrics, the “first” and “second” loops should be involved in computing SPD_m , and statements #7...#14 should be involved in computing WPD_m . So the desired code parts for obtaining parallelism metrics are found; the following step is to locate (within these parts) such statements that can be executed in parallel.

As we notice, there are no data dependencies between the statements inside the “first” loop body (Code 5.2.2). It means that all three statements (#1, #2 and #3) can be executed in parallel, i.e., in the same clock cycle. So if we perform these statements at the same time, the “first” loop body will take only 1024 cycles instead of 3072 cycles. Such reduction in the number of cycles will make the entire code to operate in FPGA about three times faster in comparison with the same code, where parallelism is not expressed. For example, if we perform the same steps with the “second” loop (Code 5.2.2), where statements #4 and #5 do not have any data dependencies and therefore can be executed in parallel, the entire code will take only $24 - (8 * 2) = 8$ cycles less (i.e, $3104 - 8 = 3096$). Such reduction in the number of cycles is so minor that it (almost) will not change the common speed performance of the entire code in FPGA. Finally, if we perform the same steps with statements #7...#14 (Code 5.2.2), which do not have any data dependencies between each other and therefore can be executed at the same time, the entire code will take only 7 cycles less (i.e, $3104 - 7 = 3095$). Such reduction in the number of cycles is so minor that it (almost) will not change the speed performance of the whole code as well.

And now the question occurs: should a designer waste his valuable design time searching data dependencies for the affinity computation and parallelism expression in such code parts, as the “second” loop with statements #7...#14 (Code 5.2.2), if FPGA cannot benefit much from them? In order to answer this question, first of all let’s calculate the given parallelism metrics (considering only source lines with statements) for Code 5.2.2. The corresponding results are shown in Table 5.2.1.

SPD	WPD	SPD_2	WPD_2
$(3+2)/14 = 0.357$	$8/14 = 0.571$	$3/14 = 0.214$	$(2+8)/14 = 0.714$

Table 5.2.1: Parallelism metric results of Code 5.2.2.

Now let’s define a global parallelism metric G (is not yet the affinity) as the sum of strong and weak parallelism degrees: $G = SPD_m + WPD_m$, and let us assume that the global metric G of 1 towards an FPGA device indicates a perfect matching, while G of 0 indicates no matching at all. So, G for the parallelism metrics, shown in Table 5.2.1, is equal to:

$$G_1 = SPD + WPD = 0.357 + 0.571 = 0.928,$$

$$G_{2_1} = SPD_2 + WPD_2 = 0.214 + 0.714 = 0.928$$

Here we notice that the values of $G_1 = G_{2_1} = 0.928$ are almost equal to 1, indicating a very high matching between FPGA and Code 5.2.2. However, at the beginning of

Section 5.2.5, it was mentioned that considering both strong and weak degrees for a global metric, the matching (provided by this metric) between an algorithm and a certain processing element may become less precise as compared with the case where only the strong degree is considered. To ensure that this is true, let's try to consider only the strong degree for the global parallelism metric. It means that in this case we should not search any data dependencies in the corresponding code parts for the estimation of weak parallelism degree, and accordingly we should not express any parallelism within those code parts for the implementation onto FPGA (i.e., we just skip those code parts). Thus, G for the metrics, shown in Table 5.2.1, is now equal to:

$$G_2 = SPD = 0.357,$$

$$G_{2_2} = SPD_2 = 0.214$$

Here we notice that G_2 and G_{2_2} differ greatly from G_1 and G_{2_1} previously calculated. At this point the following issue is faced: which of the pairs (G_1, G_{2_1}) or (G_2, G_{2_2}) provides more precise matching between FPGA and Code 5.2.2, ensuring higher execution performance of Code 5.2.2 in terms of speed? At first, it seems to us that if the values of (G_1, G_{2_1}) are much higher than those given by (G_2, G_{2_2}) , then the corresponding execution performance in case of (G_1, G_{2_1}) should be much higher than in case of (G_2, G_{2_2}) as well. However, if we believe that this is so, we will be misguided! In order to prove this assertion, first of all let's compare G_1 and G_2 with the appropriate implementation results expressed in terms of execution time, which can be calculated as

$$t_{exe} = \frac{NC}{CF}$$

where t_{exe} is the execution time of code in seconds, and NC is the number of clock cycles required to execute the code in hardware, which operates at CF clock frequency. So, with G_1 , first we express possible parallelism within the statements in the "first" and "second" loops (Code 5.2.2), involving these statements in computing SPD ; second we express possible parallelism within statements #7...#14 (Code 5.2.2), involving them in computing WPD , and finally, after parallelism is expressed, we obtain the following execution time of Code 5.2.2 (when CF , for example, is equal to 1 kHz):

$$t_{exe1} = \frac{1024 + 16 + 1}{1000} = 1.041 \text{ (sec)} \leftrightarrow G_1 = 0.928$$

With G_2 , we express possible parallelism only within the statements in the "first" and "second" loops (Code 5.2.2), involving these statements in computing SPD . So the corresponding execution time of Code 5.2.2 is now equal to:

$$t_{exe2} = \frac{1024 + 16 + 8}{1000} = 1.048 \text{ (sec)} \leftrightarrow G_2 = 0.357$$

Now let's return to the question: which of the global metrics G_1 or G_2 provides more precise matching between FPGA and Code 5.2.2, ensuring higher execution performance of Code 5.2.2 in terms of speed? As we notice, $t_{exe1} \approx t_{exe2}$, but the value of G_1 differs from the value of G_2 about three times. Since with G_2 we do not involve in the parallelism exploitation statements #7...#14, which do not gain from the use of corresponding architectural feature (i.e., inherent parallelism of FPGA), it turns out that G_2 provides much more precise matching between FPGA and Code 5.2.2. When we involve these statements (i.e., #7...#14), they greatly increase the value of a global metric G , resulting in G_1 , but (almost) do not increase the execution performance of Code 5.2.2 ($t_{exe1} \approx t_{exe2}$), or in other words, Code 5.2.2 does not benefit from the use of inherent parallelism of FPGA in statements #7...#14.

So, from the analysis made above, we have received evidence that considering both strong and weak degrees (i.e., *SPD* and *WPD*) for the global metric, the matching (provided by this metric) between the code and an FPGA device becomes less precise as compared with the case where only the strong degree is considered.

However, with G_2 , this is not always true. For example, if in Code 5.2.2 we start reducing the number of iterations both in the "first" and "second" loops (especially in the "first" loop), and start increasing the number of statements, which can be executed in parallel, outside these loops, we will obtain the opposite situation: now the appropriate code parts inside these loops will gain less from the corresponding architectural feature with the reduction in the number of loop iterations, but the code parts outside these loops will gain more from the feature with the increase in the number of appropriate statements outside the loops. We can do this until the code parts inside the "first" and "second" loops are able to gain nothing from the inherent parallelism of FPGA, and until the rest code parts (outside the loops) are able to gain a lot from this architectural feature, resulting in much higher execution performance. In this case, the value of $G_2 = SPD$ may become close to zero, indicating no matching at all. However, this possible small value of G_2 will indicate an incorrect matching between FPGA and Code 5.2.2, as the code parts (inside the "first" and "second" loops), which now do not gain from the parallelism of FPGA, are taken here in exploiting that parallelism and calculating the corresponding global metric G_2 , but the code parts (outside the loops), which can now benefit a lot from the inherent parallelism, are not taken in exploiting it and obtaining G_2 .

In order to solve this problem, which can occur without being noticed when using G_2 , we should consider the correction, made in Section 5.2.4, that is, instead of G_2 we need to use $G2_2 = SPD2$. This will distinguish between such code parts that can benefit a lot from the corresponding architectural feature and such code parts that are not able to gain much from the feature. With $G2_2$, only the code parts (i.e., highly computational loops) that can gain a lot are taken into account in obtaining the corresponding global metric (i.e., $G2_2$). In case of Code 5.2.2, the "first" loop is considered as highly computational, and therefore it is taken in calculating $G2_2$.

When using $G2_2$, we even obtain more precise matching between FPGA and Code 5.2.2 than when using G_2 , since now we do not consider for the global metric the "second" loop (Code 5.2.2), which does not gain from the inherent parallelism of FPGA as much as the "first" loop does. To make sure that in case of $G2_2$ we do not suffer from a loss in performance, let's obtain the corresponding execution time (when clock frequency $CF = 1$ kHz):

$$t_{exe3} = \frac{1024 + 24 + 8}{1000} = 1.056 \text{ (sec)} \leftrightarrow G2_2 = 0.214$$

As we notice, $t_{exe1} \approx t_{exe2} \approx t_{exe3}$. So, it appears that when using $G2_2 = SPD2$, we do not need to search data dependencies between statements for the global metric computation and parallelism expression in such code parts, as the “second” loop with statements #7...#14, since they do not benefit from the corresponding feature of FPGA (i.e., inherent parallelism).

The only problem here is that sometimes, if a code is complex, it can be difficult to find highly computational loops for $SPD2_m$. However, the appropriate steps are proposed later for that, which ensure rapid and easy distinction between highly computational loops and other code parts.

- **Bit-manipulation rate (BMR_m) metric:**

As we know from Section 5.1.2, one of the features of FPGAs is that they provide fast handling of bit-manipulation operations. It means that we can benefit from a code, which includes these operations. However, this is only valid if there are no operations different from bit-manipulation ones. In FPGAs, such operations, as division/modulo, multiplication, addition/subtraction, and shifting by a variable, produce much more complex hardware than bit-manipulation operations do. Thus, all operations here different from bit-manipulation are regarded as complex. Besides, the more complex a line of code (statement with a more complex operation) is, the longer it will take to execute in FPGA, and the lower the design clock rate will be.

Now let's consider the following C code fragment, which shows a mixture of simple and complex operations:

```
a = b & c; // AND (simple) operation
a = d ^ e; // XOR (simple) operation
a = a | e; // OR (simple) operation
a = b * c; // multiplication (complex) operation
```

The first three lines of code are simple, but the fourth is complex. The clock rate of the whole design will be limited by the fourth line. For the reason that the use of complex operations (e.g., addition/subtraction) in the given Reed-Solomon (RS) decoding algorithms is unavoidable (see the source code), the entire design is therefore limited by these operations. It means that we will not gain from the use of bit-manipulation operations in the given RS decoder. Therefore, the bit manipulation rate (BMR_m) metric for the affinity should not be considered in our case.

- **Outcome:**

After we sum up the analysis made with Code 5.2.2 and BMR_m , we can definitely state that for the best matching between an FPGA device and the given RS decoding algorithms, the best is to take only the $SPD2_m$ metric (defined in Section 5.2.4) into account in obtaining the affinity. Thus, we choose only $SPD2_m$ over another FPGA oriented metrics for the affinity.

5.2.5.2. Selection of DSP Oriented Metrics

If we take a suitable fragment of code for DSP and start analyzing it as we did with Code 5.2.2 in the previous section, we will obtain similar results, that is, if we consider both strong and weak degrees for the affinity, the matching (provided by this affinity) between an algorithm and DSP may become less precise as compared with the case where only the strong degree is considered. Accordingly, if we use SCD_{2m} , SMD_{2m} and SHD_{2m} (defined in Section 5.2.4) instead of SCD_m , SMD_m , SHD_m for the affinity calculation, we can achieve even more precise matching (see Section 5.2.5.1). Thus, we choose only SCD_{2m} , SMD_{2m} and SHD_{2m} over another DSP oriented metrics for the affinity.

5.2.5.3. Threshold for Highly Computational Loops

An additional problem here is that if we want to calculate the metrics previously selected, we need to distinguish between highly computational loops and other code parts. In order to do this, we need to define a *threshold*, which should be related to the total number of code cycles. In general, this threshold should skip over a particular amount of non-highly computational loops in a code for the affinity estimation. As we saw in Section 5.2.5.1, when we skip certain code parts, we can suffer from a loss in performance. Thus, the main purpose of our threshold is to identify how much we can lose in performance, and then to use this limit for searching highly computational loops in a code.

Now let us assume that the implementation output of RS decoder is measured in Mbit/s (bit rate), and can be calculated as

$$BR = \frac{CF \cdot BS}{NC} \quad (5.2.1)$$

where NC is the number of cycles (*latency*) required to decode one codeword of BS bits on the device's hardware, which operates at CF clock frequency, and BR is a bit rate (throughput) of the decoder.

Now let's consider that in the used RS decoding algorithm all code parts are taken into account in exploiting particular architectural features and calculating the corresponding affinities. Hence, the bit rate is equal to:

$$BR = \frac{CF \cdot BS}{NC_1} \quad (5.2.2)$$

Now let us consider that if we skip over a particular amount of non-highly computational loops in a code for the affinity estimation, we can lose, for example, one-twentieth of BR in (5.2.2), that is, if $BR = 1$ Mbit/s, we can lose $1/20 = 0.05$ Mbit/s of 1 Mbit/s, if $BR = 10$ Mbit/s, we can lose $10/20 = 0.5$ Mbit/s of 10 Mbit/s, if $BR = 100$ Mbit/s, we can lose $100/20 = 5$ Mbit/s of 100 Mbit/s, and so on. In this case, the bit rate is expressed as

$$BR - p = \frac{CF \cdot BS}{NC_2} \quad (5.2.3)$$

where $p = BR/20$ (one-twentieth of BR). As we observe, the number of clock cycles in (5.2.3) required to decode one codeword differs (is increased) from the number of cycles in (5.2.2). In order to find this difference, we need to equate (5.2.2) with (5.2.3):

$$NC_2 = \frac{NC_1 \cdot BR}{BR - p} > NC_1 \quad (5.2.4)$$

Now let's define the following proportion:

$$\begin{aligned} NC_1 & \text{ --- } 100\% \\ NC_2 & \text{ --- } X\% \end{aligned} \quad (5.2.5)$$

From (5.2.5), we obtain

$$X = \left(\frac{100 \cdot BR}{BR - p} \right) \% \quad (5.2.6)$$

So, the desired *threshold* can now be calculated as

$$Th = X \% - 100\% = \left(\frac{100 \cdot BR}{BR - p} - 100 \right) \% \quad (5.2.7)$$

where Th is the threshold in percents. It means that if a certain loop (or code part) in a given code takes less than $Th\%$ of the total number of clock cycles required to execute the code, that loop (code part) is considered as non-highly computational. Moreover, the total number of cycles required by all non-highly computational loops (code parts) should not exceed this threshold. It must also be noted that if the total number of cycles required by all non-highly computational loops (code parts) exceeds the obtained threshold, the corresponding code parts (beyond this threshold) should be interpreted as highly computational. In case of $p = BR/20$, the threshold is equal:

$$Th = \frac{100 \cdot BR}{BR - p} - 100 = \frac{100 \cdot BR}{BR - \left(\frac{BR}{20} \right)} - 100 = \frac{100}{19} = 5.26\%$$

This obtained threshold ($Th = 5.26\%$) is chosen in our case for distinction between highly computational loops and other code parts.

Nevertheless, the higher code complexity is, the more difficult it becomes to find highly computational loops (with defined threshold). In such case, different exploration tools that can quickly survey the design space and report the appropriate information are invaluable. To do this, the *Design-Trotter* tool [17] is used in our

case, which, in addition, gives an opportunity to analyze data dependencies between different statements in a given code. This tool is presented later in the current chapter.

5.2.6. The Affinity

“The affinity $A_m = [A_{GPP_m} A_{DSP_m} A_{HW_m}]$ of a method (or function) m is a triplet of values in the interval $[0, 1]$ that provides a quantification of the matching between the structural and functional features of the functionality implemented by the method and the architectural features for each one of the considered processing elements (i.e., GPP, DSP, ASIC/FPGA)” [15]. In our case, the affinity is defined as $A_m = [A_{DSP_m} A_{FPGA_m}]$, since the implementation is performed only on DSP and FPGA.

An affinity of 1 towards a processing element indicates a perfect matching, while a 0 affinity indicates no matching at all. The affinity can be expressed by a normalization function applied to a linear combination of the selected metrics, with weights that depend on the considered processing element.

In order to estimate the affinity, first of all we need to deal with the data ratio ($DR_{m,t}$) metric both for DSP and FPGA. The given DSP supports both fixed- and floating-point computation (see Appendix B), and no additional clock cycles are required for floating-point computation. With FPGAs, the use of floating-point arithmetic is inefficient, since it is more complex than integer or fixed-point arithmetic and tends to require more hardware. As follows from the above, only the $DR_{m,(float)}$ metric, which takes into account the floating-point type of data, is meaningful for the given DSP. Thus, the affinity towards the DSP device in our case involves: the *strong degrees of circularity*, *MAC* and *Harvard* (i.e., $SCD2_m$, $SMD2_m$, $SHD2_m$), and the number of variables of floating-point type (i.e., `float`). Intuitively, the affinity towards FPGA involves only the *strong degree of parallelism* ($SPD2_m$). Therefore, it is possible to evaluate the affinity for each method m as follows:

$$A_m^T = f(W \cdot C_m^T) \quad (5.2.8)$$

where

$$A_m = [A_{DSP_m} A_{FPGA_m}],$$

$$W = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

and C_m is a 5-element row vector collecting, in this order, the costs $SCD2_m$, $SHD2_m$, $SMD2_m$, $SPD2_m$, and $DR_{m,(float)}$:

$$C_m = [SCD2_m \quad SHD2_m \quad SMD2_m \quad SPD2_m \quad DR_{m,(float)}]$$

The weights of the matrix W are set to 1 when the associated metric is meaningful for a given processing element, 0 otherwise. Thus, the affinity represents the sum of all the contributions determined by each relevant metric. Since such a sum could be greater than one, the following function is applied to obtain values in the interval $[0, 1]$ allowing a direct comparison between affinity values related to different processing elements: [15]

$$f(X) = \frac{\arctan(2\pi X^2)}{\pi/2} \quad (5.2.9)$$

This normalization function is the arctangent one, as it is limited to the $[-\pi/2, \pi/2]$ interval when X varies from $-\infty$ to ∞ . In order to normalize the affinity in the interval $[0, 1]$, it is scaled of a $\pi/2$ factor. Finally, to better discriminate between low and high affinity values, a quadratic form is used in (5.2.9).

The function $f(X)$ in (5.2.9), when applied to $W \cdot C_m^T$, provides affinity values that are directly comparable. Therefore, it can be used to choose the best processing element for a given algorithm.

5.3. The Design-Trotter Tool

The Design-Trotter tool [17] is an experimental framework for guiding system designers. The main features of Design-Trotter are the characterization of the application by means of different metrics, the exploration of the application parallelism by means of dynamic trade-off curves and the possibilities of performance estimations onto existing target architectures. So, for the desired design space exploration, the following steps are performed in Design-Trotter:

1. C code to *Hierarchical Control and Data Flow Graph (HCDFG)* conversion;
2. Algorithm characterization by means of orientation and parallelism metrics;
3. Parallelism exploration.

5.3.1. C to HCDFG Conversion

The input language of Design-Trotter is a large subset of the C language. This allows us to quickly import the given RS decoding algorithms (written in C) to the tool.

In this step, the tool converts the C source code into a HCDFG (Figure 5.3.1). The HCDFG model is the internal representation of Design-Trotter and includes only HCDFGs and CDFGs. A CDFG contains only elementary conditional nodes and DFGs. A DFG contains only elementary memory and processing nodes (represents a sequence of non-conditional operations). A processing node represents a processing (arithmetic or logic) operation. A memory node represents a data-transfer (memory operation). A conditional node represents a test operation (*if*, *for*, *case*, etc.). So, actually, a certain function described in the C source code is a HCDFG. [17]

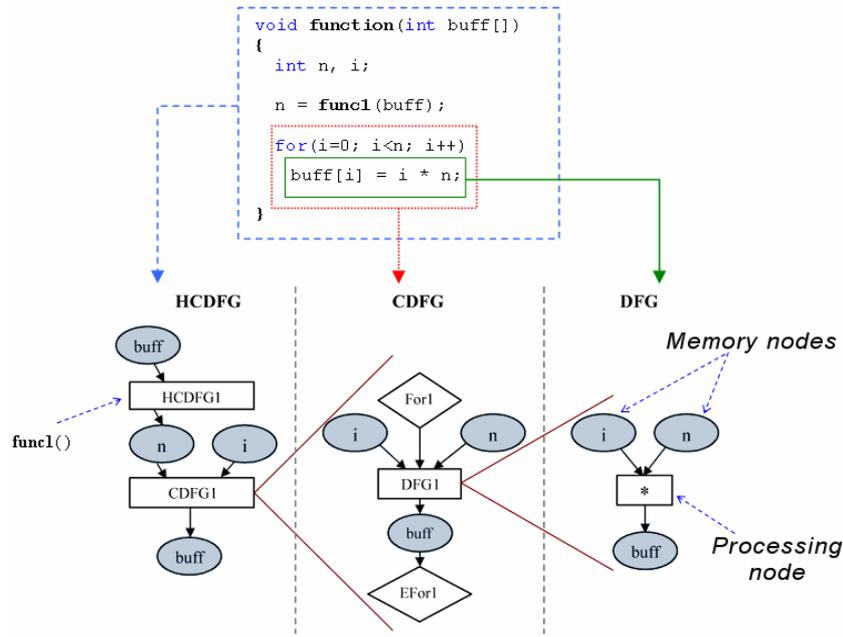


Figure 5.3.1: Example of HCDFG (with internal representation) and its corresponding C code. [18]

5.3.2. Algorithm Characterization in Design-Trotter

In this step, an abstract characterization, without any architectural assumptions, of the algorithm is performed. As we know, algorithm (or function) characterization, in general, guides the designer in his architectural choices. In Design Trotter, the characterization step has the two following objectives: [17]

1. The first one is used to sort the application functions according to their *criticality*. The criticality of a function is expressed as:

$$\gamma = \frac{NbOp}{CP} \quad (5.3.1)$$

where $NbOp$ is the sum of data-transfer plus processing operations, and CP is the *critical path* of the function, that is, the value of the longest path considering processing and data-transfer operations (i.e., the longest chain of sequential operations). This γ (gamma) metric provides a first indication about the potential parallelism of the function;

2. The second objective is to indicate the function orientation, that is, the nature of the dominant types of operations in that function. By counting tests (or control), data-transfers and processing operations within the HCDFG representation, we obtain ratios, which indicate the control and memory orientations of the function. However, algorithm characterization by means of these orientations is unusable for the selected metrics estimation, and therefore is not considered in the project.

5.3.2.1. γ Metric

For a DFG graph γ is defined in (5.3.1). γ indicates the average parallelism available in a certain function. A function with a high γ value (high parallelism) can benefit from an architecture offering high parallelism capabilities (e.g., FPGAs). From a consumption point of view, a function with a high parallelism (high γ) offers the opportunity to increase the speed performance by exploiting the spatial parallelism in hardware. On the other hand, a function with a low γ value has a rather sequential execution. For example, if $\gamma = 1$, then the function is purely sequential, and there is hardly any chance of obtaining efficient hardware implementations. In such case, it is better to use sequential processors (e.g., GPP), since they are less expensive than FPGAs.

In our case, γ is used to determine the level of parallelism in a certain function before estimating the strong parallelism degree ($SPD2_m$) metric: if $\gamma = 1$, then the function is purely sequential and $SPD2_m$ of this function will be equal to zero; therefore, no further steps are needed for $SPD2_m$ calculation. Otherwise (if $\gamma > 1$), we should perform the appropriate steps in order to obtain $SPD2_m$. So evaluating γ , we can reduce, to some extent, our design time.

Now let's obtain γ for each part of the given RS decoder (Figure 5.3.2). The corresponding results are depicted in Table 5.3.1. Actually, the RS decoder, shown in Figure 5.3.2, differs from the typical decoder, illustrated in Figure 3.5.1. The difference is that there are only three blocks instead of four (see Figure 3.5.1) in the given decoder: the Chien search in our case is combined with the Forney algorithm. This slightly reduces the memory usage in hardware and somewhat increases the execution performance in terms of speed.

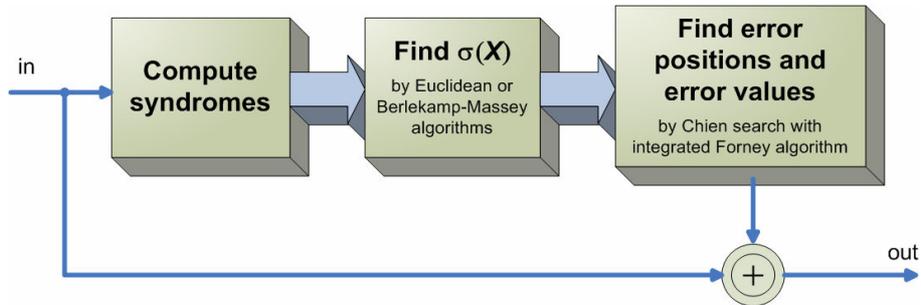


Figure 5.3.2: Block diagram of the given RS decoder with Galois field arithmetic.

	Syndrome calculation	Euclidean algorithm	Berlekamp-Massey algorithm	Chien search with integrated Forney algorithm
γ value	1.896	3.074	1.754	3.393

Table 5.3.1: Characterization of the given RS decoder by means of γ (gamma) metric.

As we see in Table 5.3.1, all γ values are greater than 1, meaning that none of the corresponding $SPD2_m$ metrics, most likely, is equal to zero. Thus, $SPD2_m$ needs to be

calculated (as described in Section 5.2.4) for each block of the given RS decoder. Moreover, in Table 5.3.1 we observe relatively high parallelism (high γ value) in the Euclidean algorithm, and in the Chien search with the integrated Forney algorithm. It means that these blocks may benefit from an architecture offering high parallelism capabilities (e.g., FPGA), resulting in higher execution performance of the entire algorithm.

5.3.3. Parallelism Exploration

In this Design-Trotter step, the exploration of the algorithm parallelism is performed. Its principle is to *schedule*, as described in the next section, the algorithm onto an *abstract architectural model* (defined in Section 5.3.5) for many time constraints. “The different time constraints express as many possible *solutions*: for a given time constraint, a certain quantity of operations has to be executed simultaneously, which implies that a sufficient quantity of operators has to be available on the architecture. The solutions generated by the Design-Trotter tool are represented with a convenient 2D graphical form using trade-off curves, which represent the number of required operators (resource usage) vs. the number of clock cycles (*cycle-budget*)” [19]. Thus, they provide estimations to evaluate the acceleration potential of a hardware implementation.

Now let’s obtain in Design-Trotter the resource vs. cycle-budget trade-off curves of the given RS decoder (Figure 5.3.2). The corresponding results are depicted in Figure 5.3.3, where ALU represents the arithmetic-logic unit that calculates arithmetic and logic operations between two numbers. So, as shown in Figure 5.3.3, the following resources are used to execute the given decoder:

- *ALU*: the maximum number of ALUs that must be used at a certain point(s) in time;
- *RAM_WRITE*: the maximum number of simultaneous memory write accesses at a certain point(s) in time;
- *RAM_READ*: the maximum number of simultaneous read accesses (at a certain point(s) in time) from memory elements that store the variable data.

Now let’s take two different solutions, for example, the first (#1) and the last (#6) from Figure 5.3.3d, where the ALU curve overlaps the RAM_WRITE curve (Table 5.3.2).

Solution #	Number of cycles	ALU	RAM_WRITE	RAM_READ
1	64112	1	1	1
6	27735	3	3	6

Table 5.3.2: Different resource usage of the first and the last solutions taken from Figure 5.3.3d.

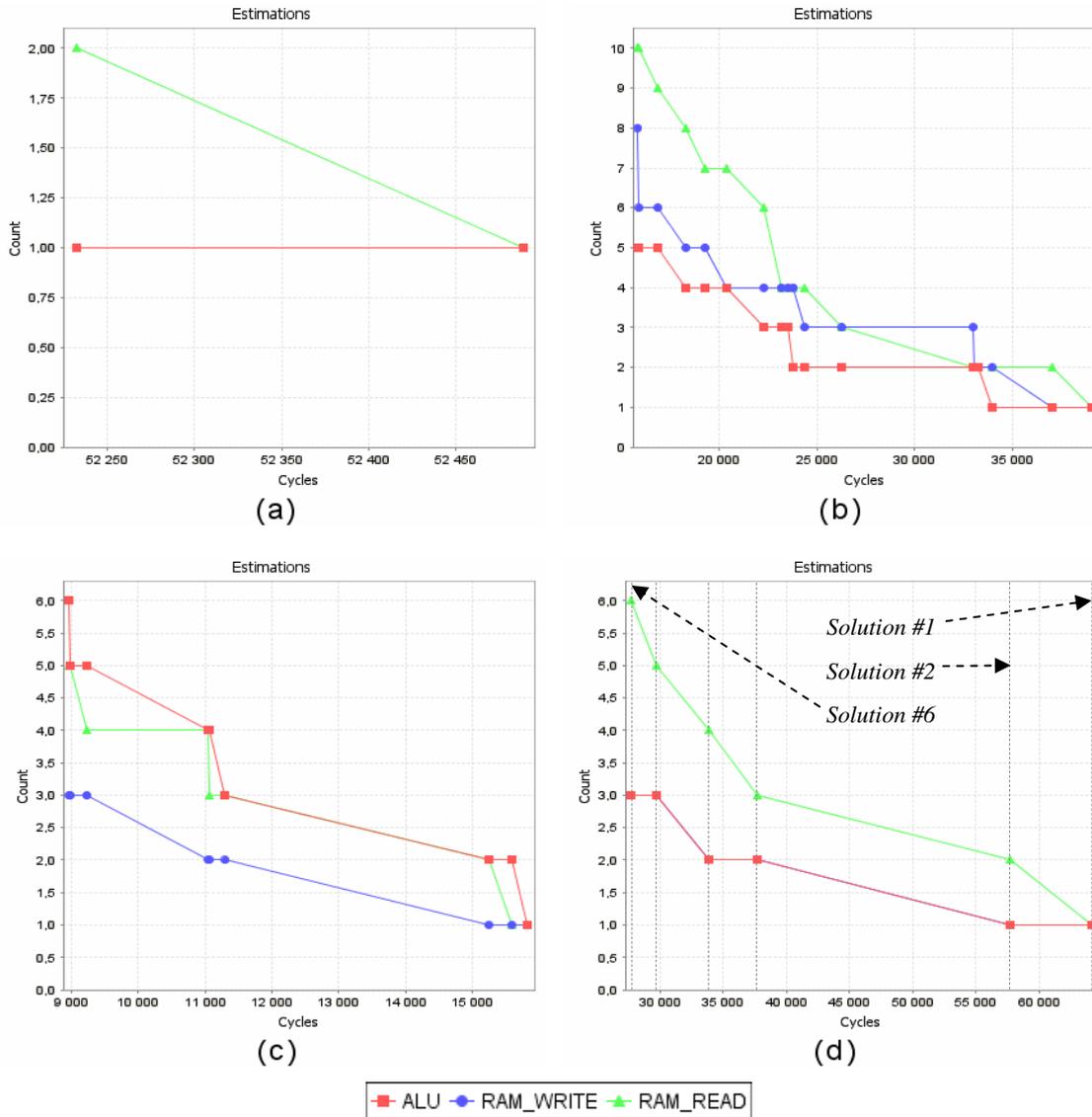


Figure 5.3.3: Resource vs. cycle-budget trade-off curves of the given RS decoder – RS(255, 239):
 (a) Syndrome calculation, (b) Euclidean algorithm, (c) Berlekamp-Massey algorithm,
 (d) Chien search with integrated Forney algorithm.

As we see both in Table 5.3.2 and Figure 5.3.3, the first taken solution is purely sequential, i.e., at this point all operations of the *error-estimation block* (Chien search with integrated Forney algorithm) are performed in a sequential manner. This takes the most time for executing the error-estimation block of the RS decoder. On the other hand, the minimum number of different operation resources is needed here. So in this case, the number of 64112 cycles (latency) is required to find the error positions and error values in a certain codeword. With the last taken solution, the maximum available parallelism for the error-estimation block is achieved, where therefore this block operates in the fastest way as compared with other solutions. So in this case, it takes only 27735 cycles. However, this correspondingly requires more resource usage at a certain time, or to be more precise, solution #6 is the most expensive in terms of resources.

5.3.4. Scheduling

The idea here is to estimate resources and bandwidth costs for several time constraints (expressed in number of cycles). At this point, Design-Trotter computes trade-off curves (defined in the previous section) by using a time-constrained scheduler that minimizes the amount of resources as well as the bandwidth, i.e., data-transfers between different memory elements. The scheduling principle is a time-constrained *list-scheduling* heuristic, where the number of resources of type i allocated at the beginning of scheduling is given by the upper bound: [20]

$$Nb\ resources_i = \frac{Nb\ operations_i}{T_C} \quad (5.3.2)$$

where T_C is the number of cycles allocated to the estimated function. Heuristics, such as list-scheduling, allow rapid computation of the resource/cycle-budget trade-off curves. However, one of the drawbacks of list-scheduling is its linear handling of the scheduling. This can lead to inaccurate resource allocation, see [20].

5.3.4.1. Schedule Details

In addition to the resource/cycle-budget trade-off curves, Design-Trotter gives an opportunity to analyze the *schedule details*. In order to see the schedule details, we click (in Design-Trotter) on the desired solution from the trade-off curves that we want to analyze; this opens a new window showing the schedule details for all the hierarchy levels of the algorithm. The information provided by the schedule details allows us to quickly determine data dependencies between different code statements and estimate cycle-budgets of desired code parts. This is very useful for the selected metrics calculation.

In our case, the purely sequential solutions, taken from the desired trade-off curves, are used to analyze cycle-budgets needed for distinction between highly computational loops and other code parts, and the solutions of maximum available parallelism are used to analyze all possible data dependencies for the calculation of strong parallelism degree ($SPD2_m$) metric, and for parallelism expression in a code.

5.3.5. Architecture Specification

Before scheduling, the designer should describe the abstract architectural model, that is, he should specify the types of operators (e.g., '+', '-', '/', and so on) to use and a number of cycles is associated to every type of operator, possibly corresponding to a given implementation device. Regarding the memory part, the designer should define the number of levels of the hierarchy and the number of cycles associated for each type of access.

With the trade-off curves, shown in Figure 5.3.3, the *system-level* is used, which permits exposure and exploration of the potential parallelism of the algorithm before choosing or building the target architecture. For that, all existing types of

processing (logic and arithmetic) operations in our case are combined in a single unit, called ALU, and only one clock cycle is associated to every type of operation (memory/processing).

5.4. The Affinity Results

In order to obtain the affinity, defined in (5.2.8), first of all we need to calculate the selected metrics. Their values for each block of the given RS decoder (Figure 5.3.2) are shown in Table 5.4.1.

	Syndrome calculation	Euclidean algorithm	Berlekamp-Massey algorithm	Chien search with integrated Forney algorithm	Entire RS decoder with Euclidean alg.	Entire RS decoder with BM ¹ alg.
$SCD2_m$	0	0.007	0.051	0	0.015	0.019
$SHD2_m$	0.077	0.131	0.051	0.056	0.108	0.057
$SMD2_m$	0	0	0	0	0	0
$SPD2_m$	0.462	0.482	0.077	0.481	0.48	0.33
$DR_{m,(float)}$	0	0	0	0	0	0

Table 5.4.1: The values of the selected metrics for the given RS decoder.

So, the affinity is now formed from Table 5.4.1, and the corresponding results are presented in Table 5.4.2. As we observe in Table 5.4.2, the A_{FPGA} values for the three following RS decoding blocks are relatively high: syndrome calculation, Euclidean algorithm, and the Chien search with the integrated Forney algorithm. It means that these blocks will benefit from an FPGA architecture, which offers high parallelism capabilities, resulting in higher execution performance (in terms of speed) of the entire algorithm. With A_{DSP} , the affinity values for all RS blocks are almost close to zero, meaning that none of these blocks will benefit much from the DSP features.

	Syndrome calculation	Euclidean algorithm	Berlekamp-Massey algorithm	Chien search with integrated Forney algorithm	Entire RS decoder with Euclidean alg.	Entire RS decoder with BM alg.
A_{DSP}	0.024	0.077	0.042	0.012	0.06	0.023
A_{FPGA}	0.591	0.617	0.024	0.617	0.616	0.382

Table 5.4.2: The affinity results for the given RS decoder.

¹ BM – Berlekamp-Massey

5.5. Summary

This chapter presented in detail the process of algorithm characterization. The characterization (affinity) results indicated that the given FPGA device is more suitable for the execution of the RS decoder than the given DSP. So the general characterization process can be summarized in the following steps:

1. First, we need to carry out an analysis of the available architectures to determine their relevant features;
2. Second, we need to define a set of appropriate metrics to identify subsets of the algorithm specification that could exploit the determined architectural features;
3. If possible, the defined metrics need to be divided into strong and weak degrees;
4. Then, we have to distinguish between code parts (in algorithm) taken into account in estimating the strong degree and code parts taken into account in estimating the weak degree;
5. In order to rapidly distinguish between strong and weak degrees: a) an appropriate threshold should be defined, and b) the Design-Trotter tool can be used to perform a subsequent analysis of an algorithm for the corresponding metrics calculation;
6. After distinction, only the strong degree must be considered for the affinity computation;
7. Finally, the corresponding metrics of strong degree should be combined to obtain the resulting affinity.

Chapter 6

FPGA IMPLEMENTATION

In Chapter 5, we performed the algorithm characterization of the given Reed-Solomon (RS) decoder. The characterization results pointed out that an FPGA device is more suitable for the execution of this decoder than DSP. In order to verify that it is true (or false), first of all the given decoding algorithms should be optimized considering the appropriate capabilities of the target architectures. Then the implementation of these algorithms should be performed both onto FPGA and DSP. And finally, for the desired verification, the corresponding implementation results should be compared with the characterization results.

The current chapter introduces the implementation process of the given RS decoder onto the given FPGA (described in Appendix A).

6.1. FPGA Design Flow

Figure 6.1.1 presents a generic FPGA design flow. The successive process phases (blocks) of Figure 6.1.1 are described as follows: [21]

- *Design Entry*: creation of design files using schematic editor or hardware description language (e.g., Verilog, VHDL). A more detailed description of hardware description languages is presented in Section 6.2;
- *Design Synthesis*: a process that starts from a high level of logic abstraction (typically Verilog or VHDL) and automatically creates a lower level of logic abstraction using a library of primitives;
- *Partition (or Mapping)*: a process assigning to each logic element a specific physical element that actually implements the logic function in a configurable device;
- *Place*: maps logic into specific locations in the target FPGA chip;
- *Route*: connections of the mapped logic;
- *Program Generation*: a user's configuration bit-stream file is generated to program the device;

- *Device Programming*: downloading the bit-stream to the FPGA;
- *Design Verification*: simulation is used to check functionalities. The simulation can be done at different levels. The functional or behavioral simulation does not take into account component or interconnection delays. The timing simulation uses *back-annotated* delay information extracted from the circuit. Other reports are generated to verify other implementation results, such as maximum frequency or delay, and resource utilization.

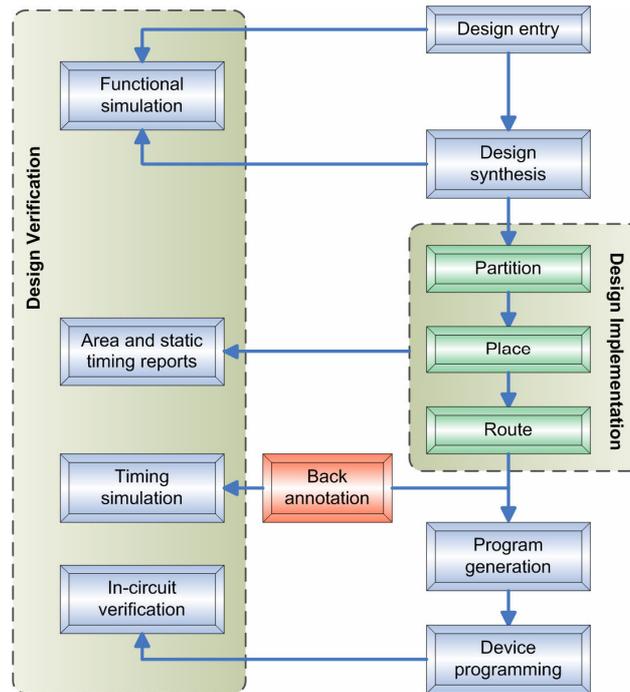


Figure 6.1.1: FPGA generic design flow.

The partition (or mapping), place, and route processes are commonly referred to as *design implementation*.

6.1.1. The ISE™ Design Flow

To apply the described above design flow to the given FPGA device, the *Integrated Software Environment (ISE™)* suite from Xilinx™ is used in the project. The ISE allows us to take our design from design entry through Xilinx device programming. The generic ISE design flow (Figure 6.1.2) comprises the following steps: [22]

- *Design Entry*. It is the first step in the ISE design flow. During design entry, we create our source files based on our design objectives using a hardware description language, such as VHDL, Verilog, or using a schematic;
- *Design Synthesis*. It is run after design entry. During this step, VHDL, Verilog, or mixed language designs become *netlist* (described in the next section) files that are accepted as input to the design implementation step;

- *Design Implementation.* After synthesis, we run design implementation, which converts the logical design into a physical file format that can be downloaded to the selected target device;
- *Xilinx Device Programming.* After generating a programming bit-stream file from the physical file, we configure our device. During configuration, we download these bit-stream files from a host computer to a Xilinx device;
- *Design Verification.* We can verify the functionality of our design at several points in the design flow. We can use *simulator software* to verify the functionality and timing of our design or a portion of the design. The simulator interprets VHDL or Verilog code into circuit functionality and displays logical results of the described hardware language to determine correct circuit operation. We can also run in-circuit verification after programming our device.

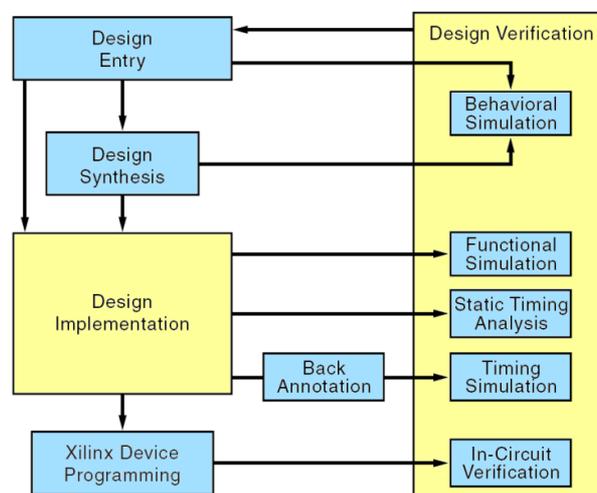


Figure 6.1.2: The ISE design flow. [22]

6.2. Hardware Description Languages

“A Hardware Description Language (HDL) is a computer language designed for formal description of electronic circuits. It can describe a circuit operation, its structure, and the input stimuli to verify the operation (using simulation). A HDL model is a text-based description of the temporal behavior and/or the structure of an electronic system. In contrast to a software programming language, the HDL syntax and semantics include explicit notations for expressing *time* and *concurrentcies*, which are the primary attributes of hardware” [21]. The two main players in this field are VHDL and Verilog. Languages, whose only characteristics are to express circuit connectivity within a hierarchy of blocks, are properly classified as *netlist* languages. One of the most popular netlist formats and industry standards is *Electronic Data Interchange Format (EDIF)*.

Traditional programming languages, such as C/C++, are sometimes used for describing electronic circuits. As these languages do not include any capability for expressing time explicitly and consequently, they are not proper hardware description languages. However, several products based on C/C++ have appeared, e.g., SystemC Handel-C.

6.2.1. Handel-C

Handel-C [23] is a “simple” programming language designed to enable the compilation of programs into synchronous, usually FPGA based, hardware implementations. Handel-C is not a hardware description language though; rather it is a programming language aimed at compiling high level algorithms into gate level hardware. Handel-C uses much of the syntax of conventional C language with the addition of inherent parallelism. Therefore it was decided to use Handel-C in the project for fast and easy transition from C-based algorithms to FPGA hardware solutions. For that, the *DK Design Suite* from Celoxica™ is used, which provides a complete design flow for implementing high-level language algorithms into hardware. Algorithms can be written directly in Handel-C, or ported from ANSI-C or C++. In the DK-suite, Handel-C code can be compiled to VHDL, Verilog, or directly to EDIF (this last format is used in our case). This allows us to use DK as the intermediate between the given RS decoding algorithms (written in C) and the ISE tools, see Figure 6.2.1.

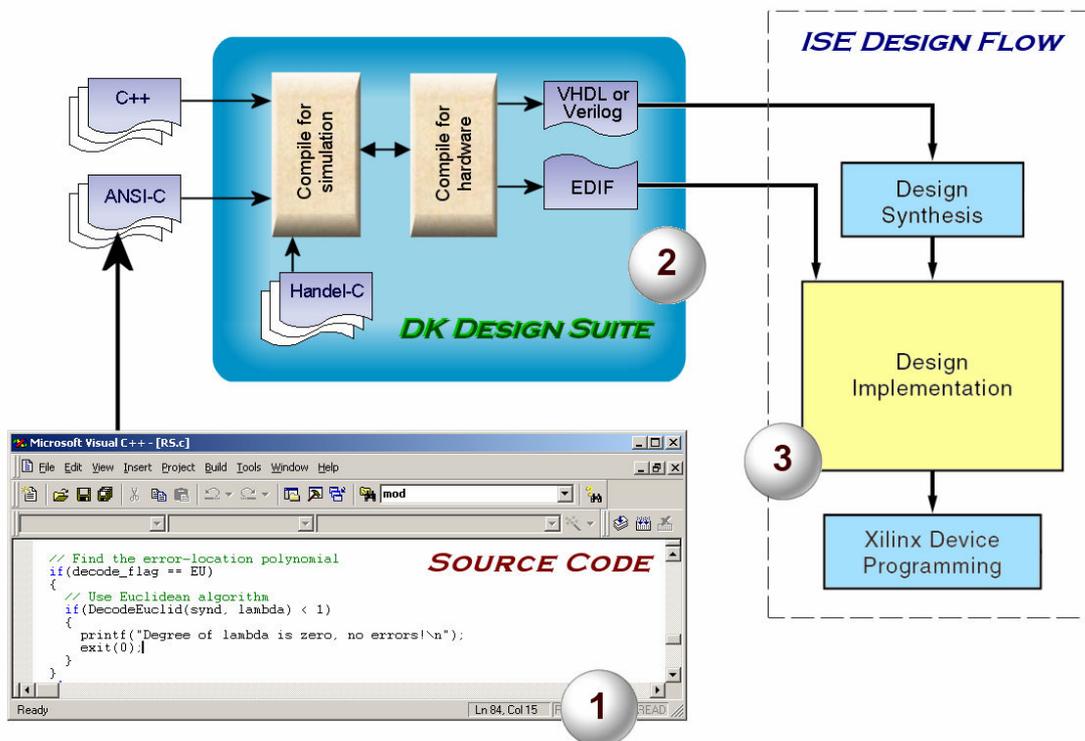


Figure 6.2.1: The design flow from C language to a complete hardware implementation.

6.2.1.1. Comparison of Handel-C and ANSI-C

Handel-C has many similarities to ANSI-C. Nevertheless, Handel-C is a language for digital logic design. This means that the way in which DK interprets it may differ to the way in which compilers interpret ANSI-C for software design. Handel-C has some extensions to ANSI-C, to allow additional functionality for hardware design. However, it also lacks some ANSI-C constructs which are not appropriate to hardware implementation.

This section summarizes only those important extensions to ANSI-C, which are used in the project. Other differences between Handel-C and ANSI-C can be found in [23].

- **Statement timing:**

When DK compiles Handel-C code for hardware implementation, it generates all the logic required to execute each line of code in a single clock cycle. Thus, the basic rule for cycles used in a Handel-C program is: *assignment (i.e., '=') takes one clock cycle.*

Some simple examples with their timings are shown below:

```
x = y; // first statement
x = (y * b) + (c * d); // second statement
```

Each of these statements takes only one clock cycle in Handel-C. However, the more complex a line of code (or statement) is, the longer it will take to execute, and the lower the design clock rate will be.

- **Parallelism:**

Handel-C implicitly executes instructions sequentially, but when targeting hardware it is extremely important to make as much use as possible of parallelism. For this reason, Handel-C also has a parallel composition keyword `par` to allow statements in a block to be executed in parallel.

The following example executes three assignment statements sequentially (it takes 3 cycles):

```
x = a;
y = b * 2;
z = c * 3;
```

In contrast, the following example executes all three assignments in parallel and in the same clock cycle:

```
par
{
  x = a;
  y = b * 2;
  z = c * 3;
}
```

With the `par` example, three specific pieces of hardware are built to perform these three assignments at the same time. It requires more amount of hardware than the corresponding sequential example.

In our case, the `par` statement presents an integral part in exploiting one of the most important features of FPGA, i.e., the high degree of inherent parallelism.

- **Arrays and memories (RAM):**

“Handel-C supports arrays used in the same way as in C, however, there are implications resulting from the way arrays are implemented in hardware. An array can be seen as a collection of variables which can all be accessed in parallel, with elements either specified explicitly or indexed by a variable. Explicit access to individual array elements is efficient, but indexing through an array can generate significant amounts of hardware, particularly if it is done from more than one point in the code”. [24]

If random access into an array cannot be avoided (this applies to our case), it is better to use a RAM instead of array, simply by adding the `ram` keyword at the start of the array declaration:

```
ram int buff[256];
```

This will create a more efficient structure in hardware, that is, RAMs are normally more efficient to implement in terms of hardware resources than arrays, but they only allow *one location* to be accessed *in any one* clock cycle.

- **Macro procedures:**

In Handel-C, placing a block of code in a function means that one copy of the code will exist in hardware, and every time the function is called, only this single copy of the code will be used, i.e., calls to functions in Handel-C result in a single shared piece of hardware. This is equivalent to an ANSI-C function, resulting in a single shared section of machine code. However, if the code block is needed to be called several times in parallel, a single function cannot be used, as multiple copies of the code are required. The way out is to use a *macro procedure* in Handel-C, or declare arrays of functions. A macro procedure builds a fresh copy of the code every time it is called. With arrays of functions, a specified number of copies is built, which can then be called in parallel. However, multiple sequential calls to a single function will result in complex circuitry at the entry and exit points of the function, leading to the following trade-offs: [24]

- A function may take up less space in hardware than a macro procedure;
- Using a macro procedure will generally result in a higher clock rate of the whole design.

The following example illustrates the use of a macro procedure:

```

macro proc output(x, y)
{
    var1 = x;
    var2 = y;
}

output(a+b, c*d); // first call
output(c*d, a+b); // second call

```

The first call writes `a+b` expression to the variable `var1`, and `c*d` to `var2`. The other call writes expressions in opposite way.

6.2.1.2. Handel-C Code Optimization

A common goal in digital hardware design is to produce circuits, which are small and run at a high clock rate. This section illustrates some important Handel-C coding styles (used in the project), which result in fast designs.

- **Complex statements:**

As was mentioned in Section 6.2.1.1, the more complex a line of code (statement) is, the longer it will take to execute, and the lower the design clock rate will be. Therefore, sometimes it is better to break a complex statement up into several simpler statements and execute them in parallel.

Now consider the following example:

```

x1 = y; // first statement
x2 = (y * b) + (c * d); // second statement

```

Each of these statements takes only one clock cycle in Handel-C. However, the clock rate of the whole design will be limited by the second statement, since it is more complex than the first one. So it would be better to break the second statement up into several simpler statements as follows:

```

par
{
    tmp1 = y * b;
    tmp2 = c * d;
}
x2 = tmp1 + tmp2;

```

Although the modified code will take three cycles to execute instead of two, this will be better overall, as the whole design will now be able to run at a higher clock rate.

- **Cycle efficiency of loops:**

“As Handel-C is very close to C, it is common to port code directly from C to Handel-C, modifying it to add parallelism. There are several areas where common coding styles in C will not produce the most efficient hardware design in Handel-C, and in the area of control statements it is the `for()` loop, which is not ideal. `for()` loops are supported by Handel-C, but because of the control portion of the loop

typically contains an assignment, it must use a clock cycle. This is because the Handel-C timing model requires every assignment to take a single clock cycle” [24]. The result is that `for()` loops have a single clock cycle overhead, so the example below takes 40 cycles to execute, rather than 20:

```
for(i=0; i<20; i++)
{
    buff[i] = 0;
}
```

To improve the performance, a `while()` loop should be used instead, as shown below:

```
i = 0;
while(i < 20)
{
    par
    {
        buff[i] = 0;
        i++;
    }
}
```

In this example the loop will now take 21 clock cycles instead of 40.

- **Timing efficient use of memories:**

As a memory of any sort includes addressing logic, there is always an inherent delay in accessing it. For that reason, a memory access is considered as a complex operation to include in a statement [24]. Thus, it is better to use an additional register for the address, and to re-use this register whenever the memory is accessed at different points in the code, as shown below:

```
MemoryAddress = addr * 5; // set up address first

par
{
    // Access memory, and set up next address
    a = Memory[MemoryAddress];
    MemoryAddress = addr + a;
}

Memory[MemoryAddress] = a; // access memory again
```

- **Parallel replicators:**

Parallel replicators can be used in Handel-C to build complex program structures quickly and allow them to be parameterized. Parallel replicators are used in the same way as `for()` loops, except that during compilation they are expanded so that all iterations are implemented individually, and can be executed in parallel. [23]

So, the following code:

```

par (i=0; i<3; i++)
{
  a[i] = b[i];
}

```

expands to:

```

par
{
  a[0] = b[0];
  a[1] = b[1];
  a[2] = b[2];
}

```

6.3. Implementation Results and Analysis

In our case, the implementation is splitted up into four parts. The implementation of each part onto FPGA is performed separately. They are described below:

1. First, the given RS decoding algorithms, written in C language, are converted to Handel-C not using any Handel-C extensions to ANSI-C, and not applying any optimization techniques, described in Section 6.2.1.2;
2. Second, in addition to the first part, the two following Handel-C extensions to ANSI-C, described in Section 6.2.1.1, are used: all possible arrays (with random access to) are replaced by RAMs, and all frequently called subfunctions are replaced by macro procedures. Moreover, the Handel-C code is now optimized, as described in Section 6.2.1.2 (only parallel replicators are not utilized for the present);
3. In the third part (is an update of the second part), the inherent parallelism of FPGA starts being exploited. At this point, parallelism is expressed within those code statements, which do not have access to the antilog and log look-up tables (described in Section 3.3.5) stored in RAMs, by using the `par` keyword in Handel-C;
4. For the reason that the look-up tables are stored in RAMs, they only allow one location to be accessed in any one clock cycle. In order to allow parallel access to the look-up tables, a specified number of their copies is built, which can then be accessed separately in parallel. So, in this final implementation part (is an update of the third part), parallelism is expressed within all possible code statements (now including access to the copies of look-up tables) by using the `par` keyword. It must also be noted that the statements, which are taken at this step into account in exploiting parallelism, they are taken as well into account in calculating the corresponding strong parallelism degree ($SPD2_m$) metric for the affinity in (5.2.8).

The results of each implementation part described above are presented for each block of the given RS decoder, shown in Figure 5.3.2. The corresponding implementation results in terms of timing performance are depicted in Table 6.3.1.

<i>Part #1:</i> <i>Original source code</i>	Syndrome calculation	Euclidean algorithm	Berlekamp-Massey algorithm	Chien search with integrated Forney algorithm	Entire RS decoder with Euclidean algorithm	Entire RS decoder with BM¹ algorithm
Latency in cycles	21198	6286	1776	41151	68635	64125
FPGA clock frequency in MHz	—	—	—	—	30.118	28.538
Bit rate of RS decoder in Mbit/s	—	—	—	—	0.84	0.85
<hr style="border-top: 1px dashed black;"/>						
<i>Part #2:</i> <i>Part #1 + Handel-C code optimization + Handel-C extensions</i>						
Latency in cycles	20945	6175	2391	36377	63497	59713
FPGA clock frequency in MHz	—	—	—	—	72.093	73.319
Bit rate of RS decoder in Mbit/s	—	—	—	—	2.17	2.35
<hr style="border-top: 1px dashed black;"/>						
<i>Part #3:</i> <i>Part #2 + expression of some parallelism</i>						
Latency in cycles	9147	3191	1812	33565	45903	44524
FPGA clock frequency in MHz	—	—	—	—	73.954	70.897
Bit rate of RS decoder in Mbit/s	—	—	—	—	3.08	3.05
<hr style="border-top: 1px dashed black;"/>						
<i>Part #4:</i> <i>Part #3 + expression of maximum available parallelism</i>						
Latency in cycles	9147	1774	1812	9798	20719	20757
FPGA clock frequency in MHz	—	—	—	—	84.774	77.604
Bit rate of RS decoder in Mbit/s	—	—	—	—	7.82	7.15

Table 6.3.1: Implementation results in terms of timing performance of the given RS decoder – RS(255, 239).

¹ BM – Berlekamp-Massey

The corresponding implementation results in terms of resource usage are shown in Table 6.3.2.

Implementation part #	Entire RS decoder with Euclidean algorithm	Entire RS decoder with Berlekamp-Massey algorithm
1	83%	75%
2	9%	6%
3	27%	18%
4	48%	37%

Table 6.3.2: Implementation results in terms of resource usage (device utilization in percents) of the given RS decoder – RS(255, 239).

As we observe both in Tables 6.3.1 and 6.3.2, the first implementation part provides the worst results in all terms: the number of cycles (latency) required to decode one codeword on FPGA is the highest, the bit rate (throughput) of the decoder is the lowest, and hardware resource (e.g., memory elements, logic gates) usage is the highest as compared to other implementation parts. With the second implementation part, where all operations of the decoder are still performed in a sequential manner, the latency is slightly reduced, and the design clock speed is greatly increased as compared to the first part. Moreover, the second part requires the lowest resource usage in comparison with other implementation parts. With the third implementation part, some parallelism is expressed in the code. This results in lower latency, but in higher resource usage as compared to the second part. Finally, with the last implementation part, the maximum available parallelism is applied. This results in the lowest latency, but in the highest hardware resource usage as compared to other implementation parts. Moreover, the fourth implementation part provides the highest clock speed, and the highest bit rate of the given decoder. Here, the achieved bit rates (7.82 Mbit/s and 7.15 Mbit/s) satisfy the ITU G.992.1 requirements in [1], which define the system to support a minimum of 6.144 Mbit/s downstream.

6.4. Summary

This chapter introduced the implementation process of the given RS decoder (written in C language) onto the given FPGA. For that, the Handel-C language was used in the project for fast and easy transition from C-based algorithms to FPGA hardware solutions. Moreover, this chapter summarized several important Handel-C extensions to ANSI-C, which were used in the project to achieve higher performance. In addition, Handel-C optimization was presented, which results in fast designs. With the actual implementation, it was splitted up into four parts. The implementation of each part onto FPGA was performed separately. The last implementation part (of maximum available parallelism) provided the highest bit rate of the decoder: 7.82 Mbit/s using Euclidean algorithm, and 7.15 Mbit/s using Berlekamp-Massey algorithm. These both bit rates satisfy the ITU G.992.1 requirements in [1].

Chapter 7

DSP IMPLEMENTATION

“*Assembly language (or assembler)* is a low-level programming language and is a more human readable form of machine language. Assembler is close to a one-to-one correspondence between symbolic instructions and executable machine codes. Programming in assembler gives direct access to key machine features essential for implementing certain kinds of low-level routines” [25]. Therefore, high quality hand crafted programs written in assembly language for DSP processors can run much faster and use much less memory and other resources than a similar program written in a high-level language (e.g. C/C++). However, assembly language is much harder to program than high-level languages, since the designer must pay attention to far more detail and must have an intimate knowledge of the DSP processor in use. Moreover, hand programming of applications in assembly language for DSPs becomes unacceptable as applications increase in complexity.

So, writing efficient assembly code for DSP architectures is a very challenging task, which slows down the development and sometimes leaves the product development team with a completely non-portable, confusing, and unmanageable source base. In order to reduce such heavy programming load, and consequently reduce the implementation time (or time-to-market), software tools, such as high-level languages and their compilers, are very important. Many programming tools and compilers are provided by vendors and researchers. With the given DSP processor, the *VisualDSP++[®] compiler* [26] is presented for the desired implementations of different applications, which can be written both in low-level (i.e., assembly) and high-level (i.e., C/C++) programming languages.

The current chapter introduces the implementation process of the given Reed-Solomon (RS) decoder onto the given DSP device (described in Appendix B) with VisualDSP++.

7.1. VisualDSP++ Environment

The VisualDSP++ environment lets programmers develop and debug applications. VisualDSP++ includes: [26]

- Integrated Development and Debugging Environment (IDDE);
- C/C++ optimizing compiler;
- Assembler and linker;
- Simulator software.

The VisualDSP++ IDDE provides complete graphical control of the edit, build, and debug process. The C/C++ compiler has been developed by the vendor for efficient translation of C/C++ code to DSP assembly. The C/C++ compiler (`ccct_s`) processes our C/C++ source files and produces TigerSHARC assembler source files. The assembler source files are assembled by the TigerSHARC assembler (`easmt_s`). The assembler creates Executable and Linkable Format (ELF) object files that can be linked (using the linker) to create an executable file. Moreover, depending on the selected target DSP processor, VisualDSP++ gives an opportunity to simulate the processor.

7.1.1. Optimizing Performance with VisualDSP++

In contrast to Handel-C in DK-suite, the C/C++ compiler in VisualDSP++ is able to automatically optimize the C/C++ code by exploiting the appropriate architectural features. Here, the compiler tries to efficiently compile the C/C++ code written in a straightforward manner to assembly code. However, at times it is difficult for the compiler to generate efficient assembly code that respects tight real-time constraints. This is mainly due to the use of DSP specific architectural features that is sometimes complicated for the compiler to exploit in a given C/C++ code. However, it is known that it is possible to improve the quality of generated assembly code by modifying (or tuning) the original C/C++ source code for the target compiler. For that, first we must understand the compiler optimizer, and must understand how to access the features of the processor. After, we can tune our application to achieve the best possible code from the compiler.

However, before optimizing we need to make sure that the given code is functional and yields correct results. One needs to realize that if a C-coded algorithm is functional and its execution speed is satisfactory, there is no need to optimize further. But if the performance of the code is not adequate, we should use the compiler optimizer. If the performance desired is still not achieved, we can tune the code for the target compiler, and then re-optimize it with this compiler. Finally, if performance is still not satisfactory, we can rewrite the time-critical sections of the code in assembly, resulting in hand-optimized code. In spite of that, the last presented optimization step is a time and cost consuming task, because it requires a thorough knowledge of both the processor architecture and the algorithm to write an efficient assembly code. Therefore, this last step is not considered in the project (according to the project design strategy, see Section 1.1). The other optimization steps are presented in the following sections.

7.1.2. Using the Compiler Optimizer

The general intention of compiler optimizations is to generate correct code that executes fast and is small in size. It must be noted that not all optimizations are suitable for every application or possible all the time. Therefore, the compiler optimizer has a number of configurations, or optimization levels, which can be applied when needed. The following list (based on [27]) identifies several optimization levels. The levels are notionally ordered with least optimization listed first and most optimization listed last:

- **Default:**

The compiler does not perform any optimization by default when none of the compiler optimization switches is enabled in VisualDSP++ project options.

- **Procedural Optimizations (PO):**

The compiler performs standard optimization on each procedure (or function) in the code being compiled. The optimizations can be directed to favor optimizations for speed or code size, or a factor between these two (`-Ov num` switch). As there is a trade-off between speed and code size, `-Ov num` switch directs the compiler to produce code that is optimized for speed versus size. The `num` variable (integer number) indicates a sliding scale between 0 and 100, which is the probability that a linear piece of generated code will be optimized for speed or for size. At `num = 0`, all code blocks are optimized for size. This is achieved by performing all optimizations except those that increase code size. At `num = 100`, all blocks are optimized for speed. At any point in between, the decision is based upon `num` and how many times a certain block is expected to be executed – the “execution count” of the block. Figure 7.1.1 demonstrates this relationship.

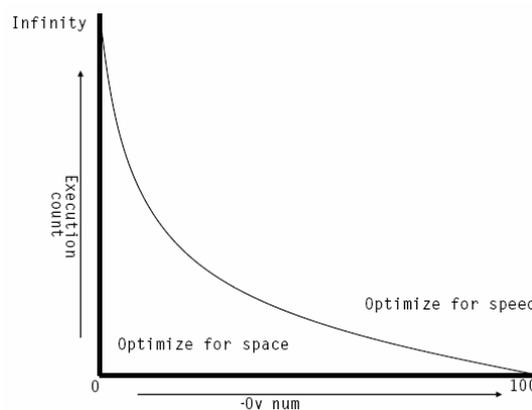


Figure 7.1.1: `-Ov` switch optimization curve. [27]

- **Profile-Guided Optimizations (PGO):**

There are many program characteristics that cannot be known statically at compile-time, but can be provided by means of PGO, which is an optimization technique that uses collected *profile* information generated from running the application to guide the compiler optimizer's decisions. The VisualDSP++ profiler determines the number of clock cycles spent executing instructions, amount of memory read or writes, etc. The compiler can use this information to achieve better optimization results. PGO should always be performed as the very last optimization step.

- **Automatic Function Inlining (AFI):**

The compiler's `inline` keyword causes a function to be "expanded" where it is called, i.e., it indicates that functions should have code generated inline at the point of call. Doing this avoids various costs, such as program flow latencies, function entry and exit instructions. So the automatic inlining switch enables the inline expansion of C/C++ functions, which are not necessarily declared inline in the source code.

The intent of performing this optimization is to improve run-time performance, at the possible cost of increasing the size of the final program. Thus, inlining has a code size-to-performance trade-off that should be considered when it is used. When AFI is enabled, the compiler automatically inlines small, frequently executed functions where possible.

- **Interprocedural Optimizations (IPO):**

IPO is a whole-program analysis. It improves performance in codes containing many frequently used functions of small or medium length. IPO tries to reduce or eliminate duplicate identical calculations, inefficient use of memory, and to simplify iterative sequences, such as loops. For example, if there is a call to another function that occurs within a loop, IPO will inline (if possible) this called function. Additionally, IPO re-orders the functions for better memory layout.

7.1.3. Tuning the Code for the Target Compiler

The main strategy for tuning a program is to present the algorithm in a way that gives the compiler optimizer excellent visibility of the operations and data, and hence a greater ability to exploit DSP specific architectural features in a given code.

This section summarizes only those tuning techniques, which are used in our case to help the optimizer better exploit the identified architectural features (described in Section 5.2.2). Other tuning techniques can be found in [27].

7.1.3.1. Quad-Word-Aligning

To make most efficient use of the hardware, it must be kept fed with data. In many algorithms, the balance of data accesses to computations is such that, to keep the hardware fully utilized, data must be fetched with loads wider than 32 bits (a word).

The given DSP device is able to load/access a quad-word data (128 bits). For that, the hardware requires that references to memory be naturally aligned. Therefore, 128-bit references must be at quad-word-aligned addresses. Thus, for the most efficient code to be generated by the compiler, we have to be ensured that data buffers are quad-word-aligned.

In general, the compiler helps to establish the alignment of array data. However, sometimes the compiler is not able to do this automatically because of complex data structures. In order to make sure that data buffers are always quad-word-aligned, we can use the appropriate compiler's pragma (`all_aligned`). By prefixing a `for` loop with this pragma, it is asserted to the compiler that every pointer variable in the loop is aligned on a quad-word boundary at the beginning of the first iteration. The following code fragment uses `all_aligned` pragma to inform compiler of alignment of `a[]` and `b[]`:

```
#pragma all_aligned
for(i=0; i<256; i++)
{
    a[i] = b[i];
}
```

7.1.3.2. Putting Arrays into Different Memory Blocks

The internal memory of the given DSP processor is divided into six blocks. Placing program instructions and data in different memory blocks, enables the DSP to support two memory operations on a single instruction line, combined with a compute instruction. However, two memory operations can be performed at the same time only if the two corresponding addresses are situated in different memory blocks (if both access the same block, then a stall will be incurred). In order to allow two memory accesses to occur simultaneously without incurring a stall, we need to put arrays into different memory blocks. This can be done by using `different_banks` pragma, which asserts to the compiler that any two independent memory accesses in the loop may be issued together without incurring a stall.

The following code fragment uses `different` memory blocks to allow simultaneous accesses to `a[]` and `b[]`:

```
#pragma different_banks
for(i=0; i<256; i++)
{
    sum += a[i] * b[i];
}
```

7.2. Implementation Results and Analysis

In our case, the implementation is splitted up into three parts. The implementation of each part onto DSP is performed separately. A short description of each part is presented below:

1. In the first part, code optimization is not used at all;
2. In the second part, the given code is tuned for the target compiler, as described in Section 7.1.3, and the procedural optimization (PO) is enabled in the compiler. Here, PO is directed to favor optimizations for code size ($num = 0$), i.e., all code blocks are optimized for size. This is achieved by performing all optimizations except those that increase code size. Those optimizations that increase code size do not exploit directly specific architectural features; they use various source code modification techniques, such as *loop unrolling* [27];
3. In order to achieve the maximum speed performance, all possible optimization techniques should be considered. So in the last implementation step, all optimizations presented before are now utilized: code tuning, PO ($num = 100$), profile-guided optimization (PGO), automatic function inlining (AFI), interprocedural optimization (IPO).

The results of each implementation part (described above) are presented for each block of the given RS decoder, shown in Figure 5.3.2. The corresponding implementation results in terms of timing performance are depicted in Table 7.2.1.

<i>Part #1:</i> <i>No optimizations</i>	Syndrome calculation	Euclidean algorithm	Berlekamp-Massey algorithm	Chien search with integrated Forney algorithm	Entire RS decoder with Euclidean algorithm	Entire RS decoder with BM ¹ algorithm
Latency in cycles	411055	92293	35154	681243	1184591	1127452
Bit rate of RS decoder in Mbit/s	—	—	—	—	0.97	1.02
<hr style="border-top: 1px dashed black;"/>						
<i>Part #2:</i> <i>Using PO and code tuning</i>						
Latency in cycles	191437	34733	16138	379163	605333	586738
Bit rate of RS decoder in Mbit/s	—	—	—	—	1.9	1.96
<hr style="border-top: 1px dashed black;"/>						
<i>Part #3:</i> <i>Using all possible optimizations: code tuning, PO, PGO, AFI, IPO</i>						
Latency in cycles	139530	17975	9750	150995	308500	300275
Bit rate of RS decoder in Mbit/s	—	—	—	—	3.72	3.82

Table 7.2.1: Implementation results in terms of timing performance of the given RS decoder – RS(255, 239). The device clock speed is constant (600 MHz).

¹ BM – Berlekamp-Massey

As we observe in Table 7.2.1, the first implementation part, where optimization is not used, provides the worst results in terms of speed performance as compared to other parts: here the bit rate of the decoder is only about 1 Mbit/s. With the second implementation part, where the source code is tuned and the procedural optimization (PO) is applied, the bit rate is increased about twice as compared to the first part. Finally, the last implementation part, where all possible optimization techniques are utilized, provides the highest bit rate in comparison with other implementation parts: the bit rate here is about twice the bit rate of the second implementation part. However, the achieved bit rates (3.72 Mbit/s and 3.82 Mbit/s) do not satisfy ITU G.992.1 [1], which defines the system to support a minimum of 6.144 Mbit/s.

7.3. DSP Implementation Results vs. FPGA Results

Now let's compare the DSP implementation results (in terms of bit rate) with the corresponding FPGA implementation results. For that, let's combine Table 7.2.1 with Table 6.3.1, resulting in Table 7.3.1.

Implementation part	DSP		FPGA	
	Bit rate in Mbit/s using Euclidean algorithm	Bit rate in Mbit/s using Berlekamp-Massey algorithm	Bit rate in Mbit/s using Euclidean algorithm	Bit rate in Mbit/s using Berlekamp-Massey algorithm
#1	0.97	1.02	0.84	0.85
#2	1.9	1.96	2.17	2.35
#3	3.72	3.82	3.08	3.05
#4	—	—	7.82	7.15

Table 7.3.1: Comparison of bit rate in DSP with bit rate in FPGA.

As can be seen in Table 7.3.1, the bit rates in the implementation parts #1, #2 and #3 of both DSP and FPGA are similar. However, the DSP device runs at 600 MHz, while the clock frequency in FPGA varies only from 29 to 85 MHz. Moreover, FPGA offers a fourth solution (implementation part #4), which is not provided by DSP.

7.4. Summary

This chapter introduced the implementation process of the RS decoder onto the given DSP device. For that, the VisualDSP++ compiler was used in the project. In addition, different optimizations (provided by VisualDSP++) were presented. With the actual implementation, it was splitted up into three parts. The last implementation part (highest optimization level) provided the highest bit rate of the decoder: 3.72 Mbit/s using Euclidean algorithm, and 3.82 Mbit/s using Berlekamp-Massey algorithm.

Chapter 8

AFFINITY RESULTS EVALUATION

After the implementation of the desired system (i.e., Reed-Solomon (RS) decoder) has been performed both onto FPGA and DSP devices, the comparison of the implementation results with the corresponding algorithm characterization results must be carried out to verify that the characterization results are true. So the current chapter goes through this verification.

8.1. Evaluation of Affinity towards FPGA

Now let's compare the obtained FPGA implementation results in terms of latency with the corresponding algorithm characterization results, i.e., with A_{FPGA} in Table 5.4.2. For that, first of all we need to define the ratio between the latency of a particular RS decoder block in the second implementation part (Table 6.3.1) and the latency of the same block in the fourth implementation part (Table 6.3.1) as follows:

$$R_{L1} = \frac{\text{Latency in cycles of block } B \text{ in part \#2}}{\text{Latency in cycles of block } B \text{ in part \#4}} \quad (8.1.1)$$

The ratio R_{L1} in (8.1.1) shows us how many times the latency of a particular decoder block, where the maximum available parallelism is expressed, is lower than the latency of the same block, where parallelism is not expressed at all. This will help us to compare the results of A_{FPGA} with the benefit derived from the use of corresponding parallelism in the code for the FPGA implementation (Table 8.1.1).

	Syndrome calculation	Euclidean algorithm	Berlekamp-Massey algorithm	Chien search with integrated Forney algorithm
A_{FPGA}	0.591	0.617	0.024	0.617
R_{L1}	2.29	3.48	1.32	3.71

Table 8.1.1: Comparison of FPGA results in terms of latency with the corresponding affinity results.

Now let's compare A_{FPGA} with R_{L1} in case of Berlekamp-Massey algorithm (Table 8.1.1). As we see, $A_{FPGA} = 0.024$, meaning that the Berlekamp-Massey algorithm is highly serial and parallelism cannot be efficiently applied here. This is confirmed by the corresponding ratio $R_{L1} = 1.32$, which indicates that the parallelism, taken into account in obtaining $A_{FPGA} = 0.024$, reduces latency of the decoder only 1.32 times in hardware. For instance, with the Euclidean algorithm, the value of $A_{FPGA} = 0.617$ is relatively high. It means that this algorithm has a regular structure and can be highly parallelized. This is confirmed by the corresponding ratio $R_{L1} = 3.48$ (Table 8.1.1), which indicates that the parallelism, taken into account in obtaining $A_{FPGA} = 0.617$, reduces latency of the decoder about 3.5 times in FPGA. In the final analysis, we can observe similar comparison results in case of syndrome calculation and Chien search with the integrated Forney algorithm.

So, after we sum up the analysis made in this section, we can state that the obtained affinity towards FPGA (Tables 5.4.2 and 8.1.1) indicates correct matching between the hardware and the given RS decoding algorithms.

8.2. Evaluation of Affinity towards DSP

Now let's compare the obtained DSP implementation results in terms of latency with the corresponding algorithm characterization results, i.e., with A_{DSP} in Table 5.4.2. For that, first of all we need to define the ratio between the latency of a particular RS decoder block in the first implementation part (Table 7.2.1) and the latency of the same block in the second implementation part (Table 7.2.1) as follows:

$$R_{L2} = \frac{\text{Latency in cycles of block } B \text{ in part \#1}}{\text{Latency in cycles of block } B \text{ in part \#2}} \quad (8.2.1)$$

The ratio R_{L2} in (8.2.1) shows us how many times the latency of a particular decoder block, which is not optimization, is higher than the latency of the same block, which is optimized by means of procedural optimization (PO) and code tuning.

Now let's define the ratio between the latency of a particular RS decoder block in the first implementation part (Table 7.2.1) and the latency of the same block in the third implementation part (Table 7.2.1) as follows:

$$R_{L3} = \frac{\text{Latency in cycles of block } B \text{ in part \#1}}{\text{Latency in cycles of block } B \text{ in part \#3}} \quad (8.2.2)$$

The ratio R_{L3} in (8.2.2) shows us how many times the latency of a particular decoder block, which is not optimization, is higher than the latency of the same block, which is optimized by means of all possible optimization techniques.

It must be noted that R_{L2} is more appropriate to comparison of the obtained DSP implementation results with the corresponding affinity results than R_{L3} , because the optimizations in R_{L3} use various source code modification techniques, such as loop unrolling [27], inlining and so on, which improve speed performance at the possible cost of increasing the size of the entire code. Therefore, these optimizations

do not exploit directly specific architectural features. With R_{L2} , the chosen level of optimizations here is more restricted within the use of various code modification techniques. The optimizations in R_{L2} are mainly related to the exploitation of the architectural features taken into account in calculating the conformable A_{DSP} . Accordingly, R_{L2} should be used to make more accurate comparison of the DSP implementation results with the corresponding affinity results (Table 8.2.1).

	Syndrome calculation	Euclidean algorithm	Berlekamp-Massey algorithm	Chien search with integrated Forney algorithm
A_{DSP}	0.024	0.077	0.042	0.012
R_{L2}	2.15	2.66	2.18	1.8
A_{DSP} / R_{L2}	0.011	0.028	0.019	0.007

Table 8.2.1: Comparison of DSP implementation results in terms of latency with the corresponding affinity results.

As we observe in Table 8.2.1, the ratios between all A_{DSP} and corresponding R_{L2} results (i.e., A_{DSP} / R_{L2}) are limited within the narrow interval. This implies that if we perform a descending (or ascending) sort on the values of each row (i.e., on A_{DSP} and R_{L2}) separately in Table 8.2.1, we will obtain an equivalent table:

	A_{DSP}	R_{L2}	A_{DSP} / R_{L2}
Euclidean algorithm	0.077	2.66	0.028
Berlekamp-Massey algorithm	0.042	2.18	0.019
Syndrome calculation	0.024	2.15	0.011
Chien search with integrated Forney algorithm	0.012	1.8	0.007



Highest value
Lowest value

Table 8.2.2: Result of a descending sort on A_{DSP} and R_{L2} values in Table 8.2.1.

Comparing Table 8.2.2 with Table 8.2.1, we notice that the relation between A_{DSP} and corresponding R_{L2} in Table 8.2.2 is exactly the same as in Table 8.2.1. It means that the obtained affinity towards DSP (Tables 5.4.2 and 8.2.1) indicates correct matching between the device and the given RS decoding algorithms.

8.3. General Affinity Evaluation

After we carried out the desired verification of the algorithm characterization results, we made sure that the affinities towards FPGA and DSP indicate correct matching between the corresponding devices and RS decoding algorithms. However, the verification was performed separately both for FPGA and DSP. This does not allow

us to easily see which type of architecture (i.e., FPGA or DSP) is the most suitable for the execution of the decoder. In order to duck the issue, we need to combine Table 8.1.1 with Table 8.2.1, resulting in Table 8.3.1, and perform a general analysis of the obtained table:

	Syndrome calculation	Euclidean algorithm	Berlekamp-Massey algorithm	Chien search with integrated Forney algorithm
A_{FPGA}	0.591	0.617	0.024	0.617
A_{DSP}	0.024	0.077	0.042	0.012
R_{L1}	2.29	3.48	1.32	3.71
R_{L2}	2.15	2.66	2.18	1.8

Table 8.3.1: Combination of Tables 8.1.1 and 8.2.1.

As we observe in Table 8.3.1, the values of A_{FPGA} greatly differ from the corresponding values of A_{DSP} (exception only with the Berlekamp-Massey algorithm), meaning that in general FPGA is more suitable for the execution of the decoder than DSP. However, when we compare R_{L1} with corresponding R_{L2} , we notice that this is not so, because the exploitation of the architectural features of both FPGA and DSP increases the decoder's performance in terms of latency about the same number of times. Therefore, at present, we may assert that the affinity results in Table 8.3.1 cannot precisely point out the most suitable type of architecture. The problem here is mostly in A_{DSP} and corresponding R_{L2} , since the chosen level of optimizations in R_{L2} exploits not only those DSP features that are taken into account in calculating the conformable A_{DSP} , but in addition to this, the optimizations in R_{L2} perform various source code modification techniques to achieve higher speed performance, see [27].

However, one important detail has been omitted. Now let's return to the FPGA implementation results in Table 6.3.1, where we observe that the design clock speed of FPGA grows with the increase in the level of expressed parallelism. It means that the exploitation of the corresponding FPGA feature (i.e., inherent parallelism) is able not only to decrease the latency, but to increase the clock frequency as well. This important detail was not taken into account for the affinity validation. In order to improve the current situation, we need to make some changes in R_{L1} and R_{L2} . The modification within R_{L1} and R_{L2} is that the ratios should now calculate not the differences between latencies, but the differences between bit rates, expressed as (5.2.1), since Expression (5.2.1) involves both the device clock speed and latency. Let's denote the modified ratios as R_{BR1} and R_{BR2} , respectively:

$$R_{BR1} = \frac{\text{Bit rate of the decoder in part \#4}}{\text{Bit rate of the decoder in part \#2}} \quad (8.3.1)$$

$$R_{BR2} = \frac{\text{Bit rate of the decoder in part \#2}}{\text{Bit rate of the decoder in part \#1}} \quad (8.3.2)$$

The ratio R_{BR1} in (8.3.1) shows us how many times the bit rate of the decoder in the fourth FPGA implementation part (Table 6.3.1), where the maximum available parallelism is expressed, is higher than the bit rate of the same decoder in the second FPGA implementation part (Table 6.3.1), where parallelism is not expressed at all. The ratio R_{BR2} in (8.3.2) shows us how many times the bit rate of the decoder in the second DSP implementation part (Table 7.2.1), where the procedural optimization (PO) and code tuning are applied, is higher than the bit rate of the same decoder in the first DSP implementation part (Table 7.2.1), where none of the presented optimization techniques is used.

Now let's compare A_{FPGA} and A_{DSP} with corresponding R_{BR1} and R_{BR2} , resulting in Table 8.3.2.

	Entire RS decoder with Euclidean algorithm	Entire RS decoder with Berlekamp-Massey algorithm
A_{FPGA}	0.616	0.382
A_{DSP}	0.06	0.023
R_{BR1}	3.6	3.04
R_{BR2}	1.96	1.92
Maximum bit rate in FPGA	7.82 Mbit/s	7.15 Mbit/s
Maximum bit rate in DSP	3.72 Mbit/s	3.82 Mbit/s

Table 8.3.2: Comparison of A_{FPGA} and A_{DSP} with corresponding R_{BR1} and R_{BR2} , and maximum bit rates of the decoder.

At last, Table 8.3.2 gladdens the eye. Now the affinity results in Table 8.3.2 are able to point out more or less precisely the most suitable type of architecture for the execution of the given RS decoder. This type is FPGA, and this is confirmed by the fact that the performance of the decoder in terms of bit rate in FPGA is about twice the bit rate of the same decoder in DSP (comparing R_{BR1} with R_{BR2} , and comparing the maximum bit rate in FPGA with the maximum bit rate in DSP).

So, after we sum up the analysis made in this section, we can affirm that the assertion made at the end of Chapter 5 that FPGA is more suitable for the execution of the decoder than DSP (before performing the actual implementation) is true.

8.4. Cost Function

Even if the algorithm characterization results towards available architectures are true, it doesn't mean that the designer should rely upon the obtained affinity and use the proposed (by this affinity) architecture for commercial activity. Maybe from a commercial point of view, the use of non-suitable (for the execution of a given

application) architecture is more beneficial in terms of market price than the use of another architecture, which was proposed by the affinity. In order to estimate this, a cost function may be defined. A cost function is usually related to different selection criteria. However, having more than one criterion, it is difficult to identify the most suitable. For that reason, we have to make a trade-off by setting up a cost function with different priorities for each criterion. But in our case, the issue can be managed without defining a particular trade-off, since there are only few parameters (criteria).

Let us suppose that the final product (i.e., RS decoder) is going to be put into mass production (e.g., thousands of units). This will compensate, to some extent, such costs, as a salary for the designer, purchasing of different development tools (e.g., DK-Suite, ISE tool, VisualDSP++), and so on. In this case, the selection of the most beneficial (in terms of market price) architecture over another should be based mainly on the following two criteria: low unit price and satisfactory time constraint.

Now let's assume that a RS decoder must follow ITU G.992.1 [1], which defines the system to support a minimum of 6.144 Mbit/s downstream. So in this case let's determine which of the given devices (i.e., FPGA or DSP) is more beneficial for commercial activity. For that, we need to consider the current market prices, which can be found, for example, in [28] (Table 8.4.1).

	FPGA	DSP
Device price	≈ \$480	≈ \$470
Maximum bit rate of the given RS decoder	7.82 Mbit/s <i>(using Euclidean algorithm)</i>	3.82 Mbit/s <i>(using Berlekamp-Massey algorithm)</i>

Table 8.4.1: Comparison of device prices with the maximum achieved bit rates of the given RS decoder.

As can be observed in Table 8.4.1, the prices of both devices are very similar, but the maximum bit rates differ about two times. Moreover, the highest bit rate in DSP (3.82 Mbit/s) does not satisfy ITU G.992.1, which define the system to support a minimum of 6.144 Mbit/s. It means that the DSP is not suitable for the commercial purpose, as it does not satisfy the given performance requirements. However, this can be improved by taking the two DSP processors for parallel performing of the two identical RS decoders. In such case, the bit rate can be increased up to $3.82 \times 2 = 7.64$ Mbit/s. This obtained bit rate now satisfies the performance requirements, and is similar to the bit rate in FPGA (7.82 Mbit/s). Nevertheless, the price of the two parallel DSPs is about twice the price of a single FPGA (Table 8.4.1). So, as follows from the above, the given FPGA should be selected for mass productions, since it cheaper than the two parallel DSPs about two times, and provides timing performance similar to that in the two DSPs ($7.82 \text{ Mbit/s} \approx 7.64 \text{ Mbit/s}$).

8.5. Summary

This chapter carried out the comparison of the DSP and FPGA implementation results with the corresponding algorithm characterization results to verify that the obtained characterization results are true. After the desired comparison, we made sure that the characterization results are true. This was confirmed by the fact that the achieved performance of the RS decoder in terms of bit rate in FPGA is about twice the maximum bit rate of the same decoder in DSP. Moreover, it was estimated that the given FPGA is more beneficial in terms of market price than the given DSP device.

Chapter 9

CONCLUSION

The main project objective was to investigate whether a fast design strategy for an efficient implementation of a Reed-Solomon decoder specified in ITU G.992.1 [1] (standard for ADSL) on the given target architectures (i.e., DSP and FPGA) can be provided or not. In order to accomplish this, the proposed design trajectory (described in Section 1.3) was evaluated.

9.1. General Summary

This section contains a summary of the main issues discussed in the report as well as the important results obtained.

- **ADSL technology:**

Digital subscriber line (DSL) technology is a home user-oriented modem technology that uses existing twisted-pair copper telephone lines to transport high-bandwidth data, such as multimedia and video. DSL service is dedicated, point-to-point, public network access over twisted-pair copper wire on the local loop between a network service provider (NSP's) central office and the customer site.

Asymmetric digital subscriber line (ADSL) is the most widely used DSL standard today. The term asymmetric reflects the difference between upstream and downstream bit rates in the transmission link. ADSL allows more bandwidth downstream – from an NSP's central office to the customer site – than upstream from the subscriber to the central office. This asymmetry, combined with always-on access, makes ADSL ideal for Internet surfing, since users typically download much more information than they send.

- **Reed-Solomon FEC in ADSL:**

The integral part of each ADSL modem is the forward error correction (FEC) technique, which is used to deliver information from a source (transmitter) to a destination (receiver) through a noisy communication channel with a minimum of

errors. FEC allows a receiver in the system to perform error detection and correction without requesting a retransmission of the corrupted data.

Reed-Solomon (RS) codes have been chosen for the FEC technique in ADSL. Here, a RS code is specified as $RS(n, k)$ with 8-bit (byte) symbols: the RS encoder in the transmitter takes k data symbols of 8 bits each and adds parity symbols (redundancy) to make an n symbol data block, called codeword. The maximum length (starting from $n = 1$) of a codeword with 8-bit symbols in ADSL is 255 bytes. There are $(n - k)$ redundant bytes. The ADSL standard requires support of all even numbers from 0 to 16 of redundancy bytes per codeword. This would allow for up to 8 bytes to be in error for every RS codeword. The RS decoder at the receiver removes the redundancy introduced by the RS encoder at the transmitter, and attempts to detect and correct possible bit errors using the knowledge of the code used by the channel encoder and the redundancy contained in the received data.

- **System simulation:**

Before a real-time implementation was initiated, it was very helpful to perform a simulation of the given system (i.e., RS decoder): after the system was simulated and the practical performance was obtained, it was necessary to compare the obtained performance with the theoretical one to ensure that the system at hand works correctly. Besides, according to the system constraints described in Section 1.4, and according to the system requirements described in Section 1.5, the $RS(255, 239)$ code was selected over another from the simulation results for its further analysis and implementation on the target architectures.

The $RS(255, 239)$ code provides the best bit-error performance, but it is the most complex code in comparison with the other supported by ADSL RS codes, as the codeword size and redundancy of $RS(255, 239)$ are the maximum available in ADSL. This certainly leads to high implementation complexity. However, from the corresponding analysis it was found out that the $RS(255, 239)$ code takes several advantages for ADSL: 1) if SNR per bit (E_b/N_0), at which $RS(255, 239)$ provides the best BER, is satisfied, Trellis coding in ADSL can be eliminated; 2) the CRC operation in an ADSL modem can be eliminated as well.

- **Algorithm characterization:**

According to the proposed design trajectory, it was necessary to determine (before the implementation step) which type of architecture (DSP or FPGA) is the most suitable for the execution of the given RS decoder. For that, algorithm characterization was performed. The main idea of characterization is to extract relevant information from the given algorithm to guide the designer towards an efficient algorithm-architecture matching. For this purpose, different performance metrics were efficiently used in the project to rapidly stress the proper architecture style for the RS decoding algorithms.

Performance metrics are the mean of evaluating a given design specification to test its particular properties. These metrics can be properly combined in order to build a global metric (the affinity) able to suggest the most suitable type of architecture for the execution of an algorithm. To obtain the affinity for the given RS decoder, the following two subtasks were performed:

1. First, an analysis of the available architectures was carried out to determine their relevant features. With the given DSP, the following architectural features of such processing element were identified: circular addressing, multiply and accumulate (MAC) operations, and Harvard architecture. With FPGA, the following two features were considered: inherent parallelism and fast handling of bit-manipulation operations;
2. Second, a set of appropriate metrics was defined to identify subsets of the algorithm specification that could exploit the determined architectural features. Then, the metrics were classified into two groups: DSP oriented and FPGA oriented. The purpose of DSP oriented metrics is to identify functionalities suitable to be executed by a DSP device by considering those issues that exploit the most relevant architectural features of such processing element. The goal of FPGA oriented metrics is to highlight relevant FPGA features.

In our case, some of the defined metrics were divided into strong and weak degrees. The difference between these two degrees is that certain source code parts taken into account in calculating the strong degree of a particular architectural feature can benefit more from that feature than certain code parts taken into account in calculating the corresponding metric of weak degree. In order to rapidly distinguish between code parts taken into account in estimating the strong degree and code parts taken into account in estimating the weak degree, the Design-Trotter tool was used in the project.

So, once the metrics have been defined, they have to be taken into account in defining the affinity, which towards a certain processing element depends on the degree (strong and weak) of a particular set of metrics. From the corresponding analysis it was found out that considering both strong and weak degrees for the affinity, the matching (provided by this affinity) between an algorithm and a certain processing element may become less precise as compared with the case where only the strong degree is considered. For that reason, only the strong degree of metrics was chosen for the affinity computation.

- **Affinity evaluation:**

After a proper combination of the selected metrics, the corresponding affinity was obtained. The affinity results pointed out that an FPGA device is more suitable for the execution of the given RS decoder than DSP. In order to verify that it is true (or false), first of all the decoding algorithms were optimized considering the appropriate capabilities of the target architectures. Then the implementation of these algorithms was performed both onto FPGA and DSP. And finally, for the desired verification, the corresponding implementation results were compared with the algorithm characterization (affinity) results. After this comparison, we made sure that the characterization results are true. This was confirmed by the fact that the maximum achieved performance of the decoder in terms of bit rate in FPGA is about twice the maximum bit rate of the same decoder in DSP (referring to the implementation results).

- **Implementation results:**

The maximum achieved bit rate of the RS decoder in FPGA is 7.82 Mbit/s (using Euclidean algorithm), and the maximum achieved bit rate of the decoder in DSP is 3.82 Mbit/s (using Berlekamp-Massey algorithm). It must also be noted that the achieved bit rate in FPGA (i.e., 7.82 Mbit/s) satisfied the ITU G.992.1 requirements in [1], which define the system to support a minimum of 6.144 Mbit/s downstream. However, with the given DSP, this requirement is not satisfied.

- **Outcome:**

So, in the last analysis, we can affirm that the proposed design trajectory was successfully applied. This trajectory has presented a rapid process of an efficient implementation of the RS decoder onto the available architectures (DSP and FPGA).

9.2. Applying the Proposed Design Trajectory to Other Types of Applications

This section presents the main steps of a rapid design strategy for an efficient implementation of an algorithm(s) in use considering available types of architectures (e.g., GPP, DSP, FPGA). For that, the proposed design trajectory (described in Section 1.3) is changed to a generic form (Figure 9.2.1). This generic form is described below:

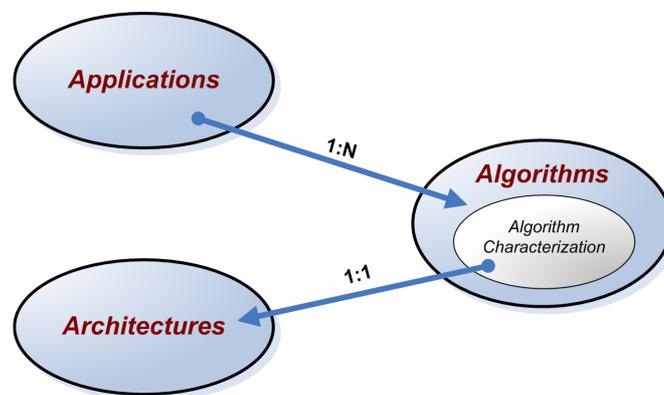


Figure 9.2.1: A generic form of the design trajectory, described in Section 1.3.

- **Applications domain:**

This domain should be used mostly for:

1. Specifying the system;
2. System analysis;
3. Defining the main tasks that the system must perform.

- **Algorithms domain:**

This domain should be used for:

1. Algorithm(s) development and simulation;
2. Algorithm characterization.

From the simulation results we need to select only those algorithms which satisfy the appropriate application requirements. After that, the characterization should be performed (as described in Chapter 5) on the selected algorithms. From the obtained characterization (affinity) results, we have to find the highest affinity value. This highest value will correspond to a particular algorithm, and to a particular type of architecture.

- **Architectures domain:**

Finally, we implement the algorithm proposed by the affinity onto the corresponding architecture, which is proposed by this affinity as well.

Appendix A

PROGRAMMABLE LOGIC

All *logic devices* can be classified into two broad categories: *fixed* and *programmable*. “Circuits in a fixed logic device are permanent: they perform one function or set of functions, and once manufactured, they cannot be changed. With the *Programmable Logic Devices (PLDs)*, ready-made parts can be modified at any time to perform any number of functions. A key benefit of using PLDs is that, during the design phase, designers can change the circuitry as often as they want until the design operates satisfactorily. PLDs are based on rewritable memory technology: to modify the design, the device only needs to be reprogrammed. Reusability is a further attractive feature of PLDs. Within programmable logic devices, two major types deserve to be highlighted: the *Complex Programmable Logic Device (CPLD)* and *Field Programmable Gate Array (FPGA)*” [29]. They are described below.

Moreover, the particular FPGA device (i.e., Xilinx Virtex-II) used in the project is described in this appendix (i.e., in Section A.3).

A.1. Programmable Logic Devices (PLDs)

The original programmable logic devices (PLDs) were the first chips that could be used as hardware implementation of a flexible digital logic design. “A PLD is made of a fully connected set of *macrocells*. These macrocells typically consist of some combinational logic (i.e., AND/OR gates and a flip-flop), see Figure A.1.1. A small Boolean equation can thus be built within each macrocell. This equation will convert the state of some binary inputs into a binary output and, if necessary, store that output in a flip-flop until the next clock edge”. [29]

“As chip densities increased, PLD manufacturers naturally developed their products toward larger parts, called *Complex Programmable Logic Devices (CPLDs)*. In a certain respect, CPLDs can be described as several PLDs (plus some programmable interconnection) in a single chip. The larger size of a CPLD allows implementing either more logic equations or more complicated designs” [29]. A block diagram of a typical CPLD is shown in Figure A.1.2, where each *logic block* is equivalent to one PLD. Because CPLDs can hold larger designs than PLDs, their potential uses are quite wide-ranging.

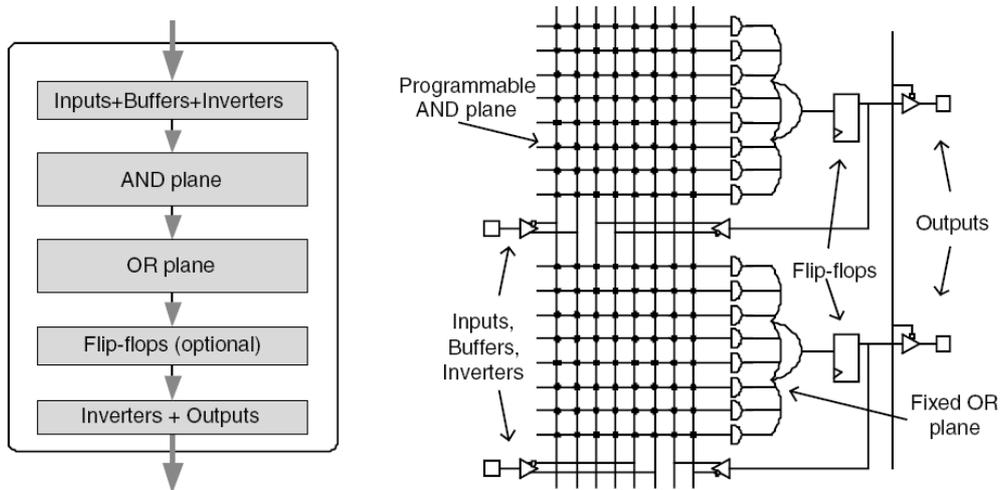


Figure A.1.1: Typical PLD architecture. [29]

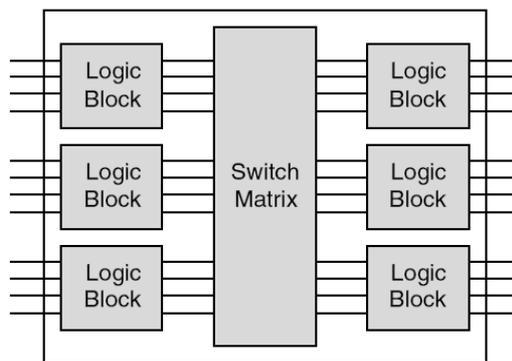


Figure A.1.2: Internal structure of a theoretical CPLD. [29]

CPLDs are based on one of three process technologies:

- Erasable Programmable Read-Only Memory (EPROM);
- Electrically Erasable Programmable Read-Only Memory (EEPROM), or
- FLASH memory.

EPROM-based CPLDs are usually *One-Time Programmable (OTP)*. Once programmed, an EPROM can be erased only by exposing it to strong ultraviolet light. With EEPROM and FLASH, they can be programmed and erased electrically. However, not all EEPROM- and FLASH-based devices are programmable while soldered on a board. [30]

A.2. Basic FPGA Concepts

A field-programmable gate array (FPGA) is a large-scale integrated circuit that can be programmed after it is manufactured rather than being limited to a predetermined, unchangeable hardware function. The term "field-programmable" refers to the ability to change the operation of the device "in the field," while "gate array" is a somewhat dated reference to the basic internal architecture that makes this after-the-fact reprogramming possible. These FPGAs can be used to implement just about any hardware design.

“The basic FPGA architecture consists of a two-dimensional array of programmable logic blocks and flip-flops with means for the user to configure: a) the function of each logic blocks, b) the inputs/outputs (I/O), and c) the interconnection between blocks (Figure A.2.1). Families of FPGAs differ from each other by the physical means for implementing user programmability, arrangement of interconnection wires, and basic functionality of the logic blocks”. [29]

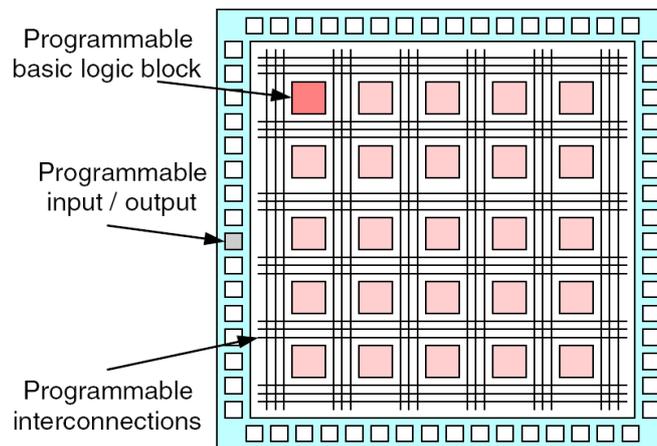


Figure A.2.1: Basic architecture of FPGA. [29]

A.2.1. Programming Methods

The information presented in this section is extracted from [29].

There are three main types of programmability:

- *Static Random Access Memory (SRAM) Based* (e.g., Xilinx™, Altera™): FPGA connections are achieved using pass-transistors, transmission gates, or multiplexers (MUXs) that are controlled by SRAM cells (Figure A.2.2a). This technology allows fast in-circuit reconfiguration. The major disadvantages are the size of the chip, required by the SRAM technology, and the needs of some external source (usually external nonvolatile memory chips) to load the chip configuration. The FPGA can be programmed an unlimited number of times;

- *Anti-fuse Technology* (e.g., Actel™, Quicklogic™): an anti-fuse remains in a *high-impedance state* (i.e., the wire is not being driven to a logical high or a logical low) until it is programmed into a low-impedance or “fused” state (Figure A.2.2b). This technology can be used only once on OTP devices. It is less expensive than the SRAM technology;
- *EPROM/EEPROM Technology* (various PLDs): this method is the same as that used in EPROM/EEPROM memories. The configuration is stored within the device, that is, without external memory. Generally, in-circuit reprogramming is not possible.

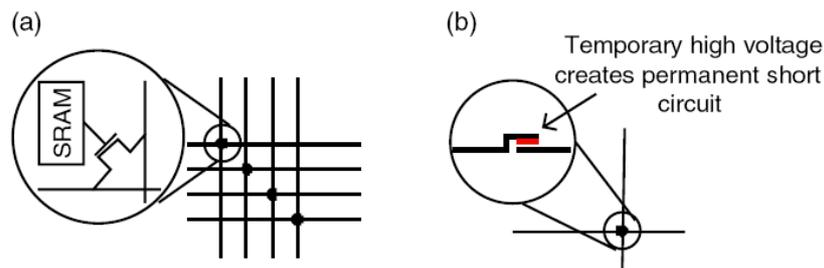


Figure A.2.2: Programming methods: a) SRAM connection; b) Anti-fuse.

A.2.2. Look-Up Tables

The way logic functions are implemented in a FPGA is another key feature. Logic blocks that carry out logical functions are usually *Look-Up Tables (LUTs)*, implemented as memory, or multiplexer and memory (Figure A.2.3). A $2^n \times 1$ memory can implement any n -bit function. Typical sizes for n are 2, 3, 4, or 5.

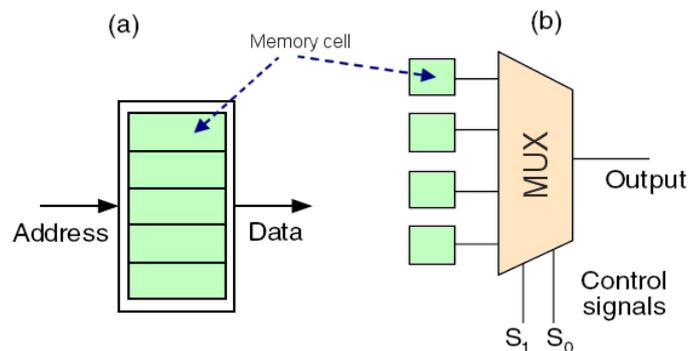


Figure A.2.3: Look-up table implemented as: a) memory, or b) multiplexer and memory. [29]

In Figure A.2.3a, an n -bit LUT is implemented as a $2^n \times 1$ memory: the input address selects one of 2^n memory locations. The memory locations (e.g., SRAM cells) are normally loaded with values from the user’s configuration bit-stream.

In Figure A.2.3b, the multiplexer control inputs are the LUT inputs. The result is a general-purpose “logic gate.” An n -LUT can implement any n -bit function. An n -LUT is a direct implementation of a function *truth table*. Each memory cell holds the value of the function corresponding to one input combination. An example of a 3-LUT is shown in Figure A.2.4.

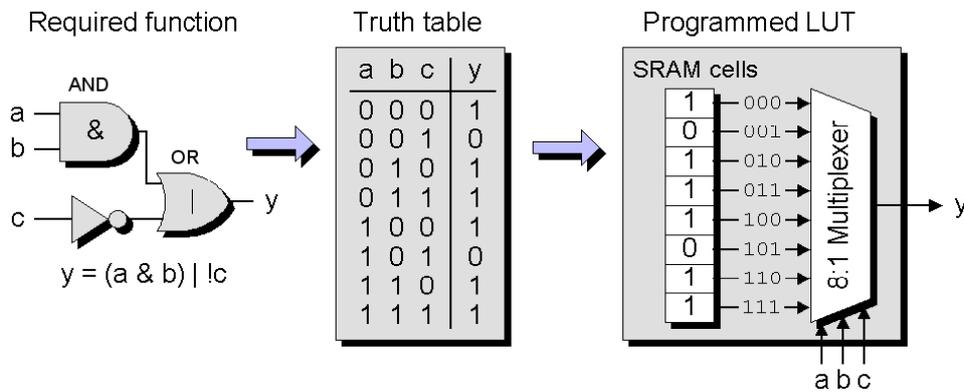


Figure A.2.4: Configuring a LUT, which is required to perform the function: $y = (a \& b) | c$. This is achieved by loading the 3-LUT with the appropriate function output values. [30]

A.2.3. FPGA Logic Block

There are two fundamental incarnations of the programmable logic blocks: *MUX-based* and *LUT-based*:

- **MUX-based:**

As an example of a MUX-based approach, consider one way in which the 3-input function

$$y = (a \& b) | c$$

could be implemented using a block containing only multiplexers (Figure A.2.5). The device can be programmed such that each input to the block is presented with a logic 0, a logic 1, or the true or inverse version of a signal (a , b , or c in this case) coming from another block or from a primary input to the device. This allows each block to be configured in myriad ways to implement a plethora of possible functions.

- **LUT-based:**

A simplified FPGA logic block can be designed with a LUT (typically a 4-input LUT), implementing a combinational logic function, and a register (flip-flop) that optionally stores the output of the logic generator (LUT), see Figure A.2.6. The number of LUTs in the logic block usually ranges from 2 to 4. The idea of programming these LUTs is shown in Figure A.2.4.

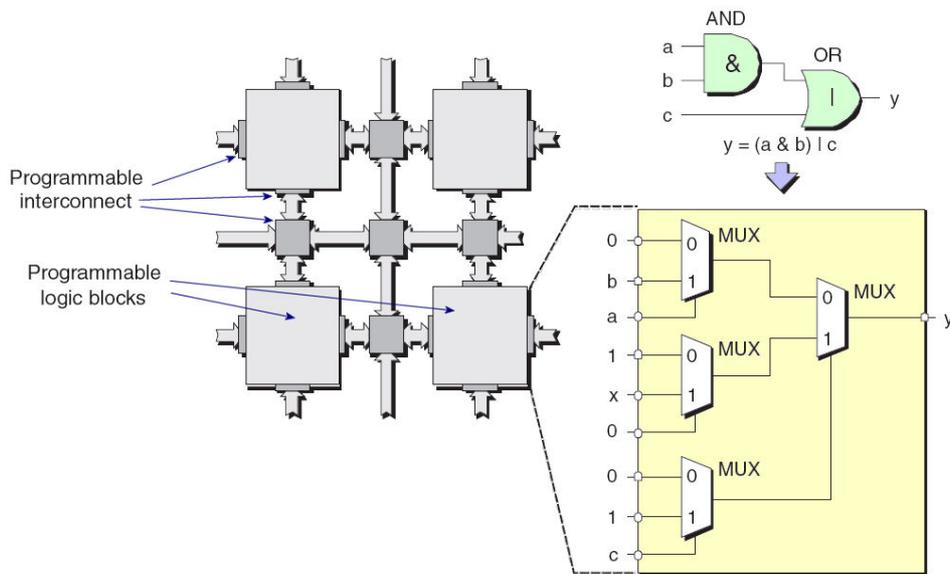


Figure A.2.5: MUX-based logic block. The x shown on the input to the central multiplexer indicates that we don't care whether this input is connected to a 0 or a 1. [30]

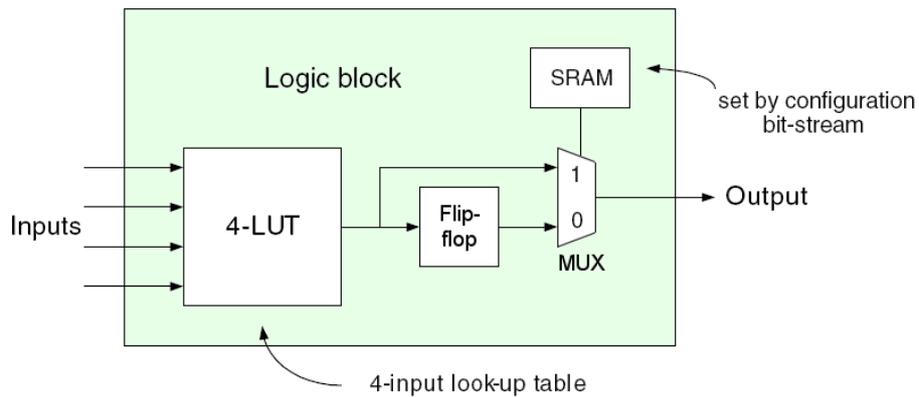


Figure A.2.6: LUT-based logic block. [29]

A.3. Xilinx™ Specifics

This section is devoted to the description of the *Xilinx Virtex-II* device family, since this FPGA is used in the project. Virtex-II devices contain the following resources (Figure A.3.1):

- Configurable Logic Blocks (CLBs);
- Input/Output Blocks (IOBs);
- RAM blocks;
- Dedicated multipliers;

- Programmable Interconnections (PIs);
- Digital Clock Manager (DCM);
- Other resources: three-state buffers, global clock buffers, and so on.

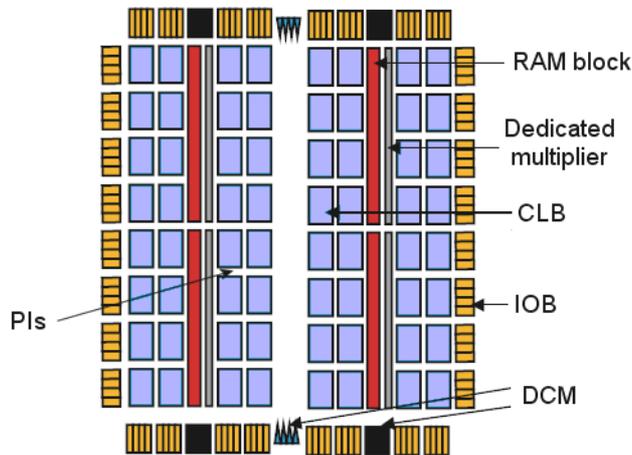


Figure A.3.1: Example of distribution of CLBs, IOBs, PIs, RAM blocks, DCMs, multipliers in Virtex-II device.

A.3.1. Configurable Logic Blocks

The core building block in a modern FPGA (the same for Virtex-II) from Xilinx is called a *logic cell (LC)*. An LC comprises a 4-input LUT (which also acts as a 16×1 RAM or a 16-bit shift register), a multiplexer, and a register, which can be configured to act as a flip-flop, as shown in Figure A.3.2.

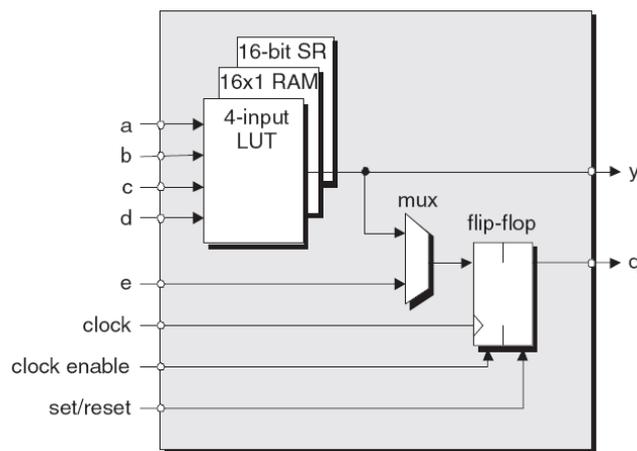


Figure A.3.2: A simplified view of a Xilinx LC. [30]

In addition to the LUT, MUX, and register, the LC also contains a smattering of other elements, including some special *fast carry logic* for use in arithmetic operations.

This special carry logic boosts the performance of logical functions such as counters and arithmetic functions such as adders.

The next step up the hierarchy is what Xilinx calls a *slice*. A slice contains two logic cells (Figure A.3.3). With the slice, each logic cell's LUT, MUX, and register have their own data inputs and outputs; the slice has one set of *clock*, *clock enable*, and *set/reset* signals common to both logic cells.

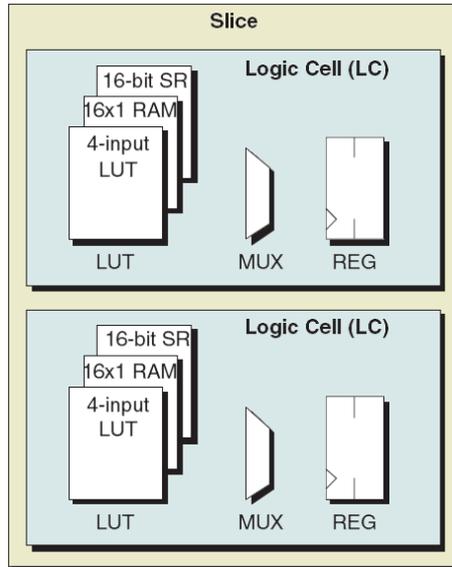


Figure A.3.3: A slice containing two logic cells. [30]

And moving one more level up the hierarchy, we come to what Xilinx calls a *configurable logic block (CLB)*. Depending on the FPGA family, CLBs contain different number of slices. Virtex-II FPGA holds four slices per CLB (Figure A.3.4).

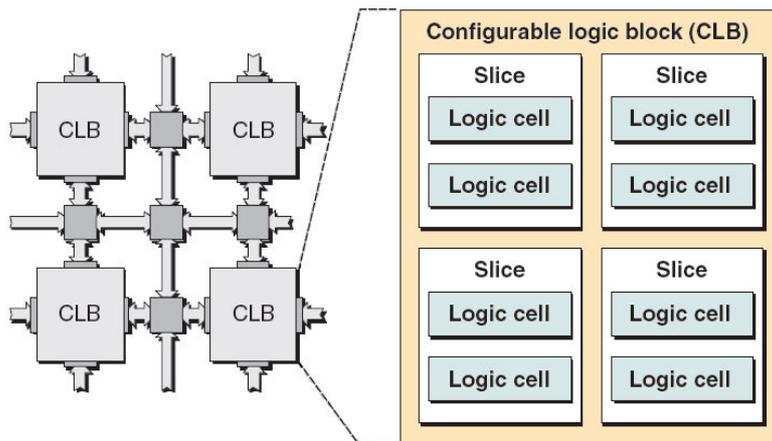


Figure A.3.4: A CLB containing four slices. The fast programmable interconnect is used to connect neighboring slices within the CLB. [30]

So a CLB equates to a single logic block in our original visualization of “islands” of programmable logic in a “sea” of programmable interconnect.

A.3.1.1. Distributed RAMs and Shift Registers

Previously, it was noted that each 4-bit LUT can be used as a 16×1 RAM. So all of the LUTs within the CLB, shown in Figure A.3.4, can be configured together to implement the following:

- Single-port (16×8)-bit SRAM;
- Single-port (32×4)-bit SRAM;
- Single-port (64×2)-bit SRAM;
- Single-port (128×1)-bit SRAM;
- Dual-port (16×4)-bit SRAM;
- Dual-port (32×2)-bit SRAM;
- Dual-port (64×1)-bit SRAM.

“Alternatively, each 4-bit LUT can be used as a 16-bit shift register. In this case, there are special dedicated connections between the logic cells within a slice and between the slices themselves. This allows the LUTs within a single CLB to be configured together to implement a shift register containing up to 128 bits as required”. [30]

A.3.2. RAM Blocks

A lot of applications require the use of memory, so Xilinx FPGA includes relatively large chunks of embedded (or dedicated) RAM, called *RAM blocks*. These memory blocks are organized in columns along the chip (Figure A.3.1). In Virtex-II, each block is a fully synchronous dual-ported 18-kbit RAM, with independent control signals for each port. The data width of the two ports can be configured independently. Moreover, each block of RAM can be used independently, or multiple blocks can be combined together to implement larger memory blocks. There are 96 RAM blocks in Virtex-II. [31]

A.3.3. Dedicated Multipliers

Some functions, like multipliers, are inherently slow if they are implemented by connecting a large number of programmable logic blocks together. Since these functions are required by a lot of applications, many FPGAs incorporate special hard-wired (or dedicated) multiplier blocks. With Xilinx, these are located in close proximity to the embedded RAM blocks (Figure A.3.1).

In Virtex-II, the dedicated multiplier block is a (18 x 18)-bit multiplier and is optimized for operations based on the RAM block content on one port. The 18 x 18

multiplier can be used independently of the RAM block resource. There are 96 dedicated multipliers in Virtex-II. [31]

A.3.4. Input/Output Blocks

The *input/output blocks (IOBs)* provide the interface between the FPGA and the outside world (Figure A.3.1). The Xilinx IOB includes inputs and outputs that support a wide variety of I/O signaling standards. IOBs have storage elements that act as registers (flip-flops). IOBs are programmable and can be categorized as follows: [29]

- *Input Path:* A buffer in the IOB input path is routing the input signals either directly to internal logic or through an optional input flip-flop;
- *Output Path:* The output path includes a three-state output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer directly from the internal logic or through an optional IOB output flip-flop;
- *Bidirectional Block:* This can be any combination of input and output configurations.

A.3.5. Digital Clock Manager

“All of the synchronous elements inside an FPGA (e.g., registers inside the CLBs) need to be driven by a clock signal. Such a clock signal typically originates in the outside world, comes into the FPGA via a special clock input pin, and is then routed through the device and connected to the appropriate components”. [30]

“Consider a simplified representation that omits the programmable logic blocks and shows only the *clock tree* and the registers (configured to act as flip-flops) to which it is connected (Figure A.3.5)”. [30]

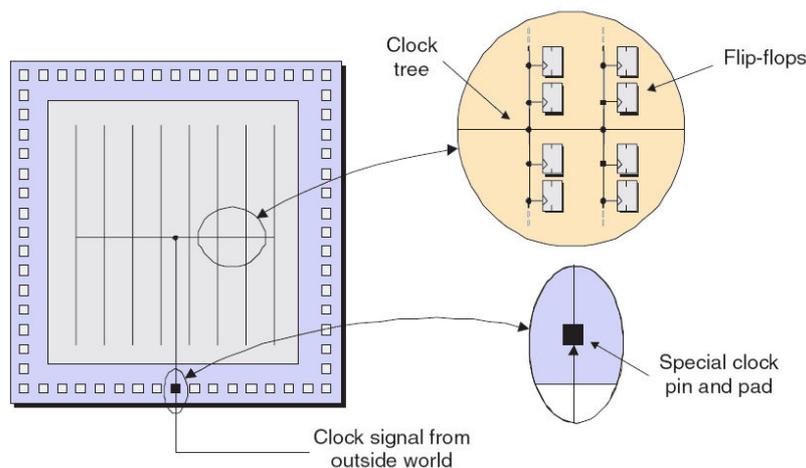


Figure A.3.5: A simple clock tree. [30]

“This is called a *clock tree* because the main clock signal branches again and again (the flip-flops can be considered as the “leaves” on the end of the branches). This structure is used to ensure that all of the flip-flops see their versions of the clock signal as close together as possible” [30]. In reality, multiple clock pins are available, and thus there are multiple *clock domains* (clock trees) inside the device.

“Instead of configuring a clock pin to connect directly into an internal clock tree, that pin can be used to drive a special hard-wired block, called a *clock manager*, which generates a number of *daughter clocks* (Figure A.3.6)”. [30]

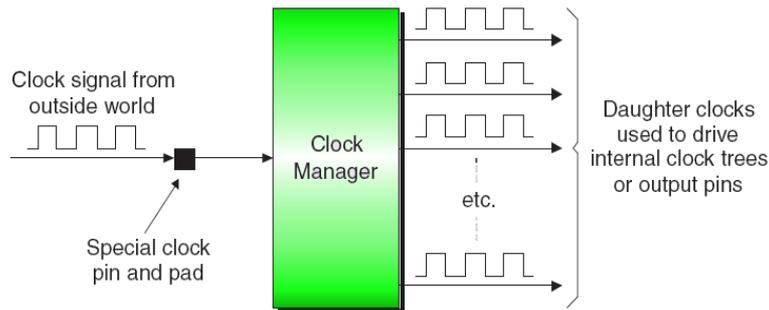


Figure A.3.6: A clock manager generates daughter clocks. [30]

These daughter clocks may be used to drive internal clock trees or external output pins that can be used to provide clocking services to other devices on the host circuit board.

Each family of FPGAs has its own type of clock manager. In the Xilinx world, a clock manager as described here is referred to as a *digital clock manager (DCM)* (Figure A.3.1). In Virtex-II, up to 12 DCM blocks are available. [31]

A.3.6. Programmable Routing

So, the internal configurable logic of FPGA includes four major elements organized in a regular array: CLBs, RAM blocks, dedicated multipliers, DCM blocks. All of these elements are interconnected by special programmable routing resources, called *General Routing Matrix (GRM)*. The GRM is an array of routing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the GRM (Figure A.3.7). The GRM is controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded. The overall programmable interconnection is hierarchical. [31]

Most Virtex-II signals are routed using the global routing resources, which are located in horizontal and vertical routing channels between each switch matrix. Horizontal and vertical routing resources for each row or column include (Figure A.3.8): [31]

- *Long Lines*: bidirectional wires that distribute signals across the device. Vertical and horizontal long lines span the full height and width of the device;

- *Hex Lines* route signals to every third or sixth block away in all four directions;
- *Double Lines*: route signals to every first or second block away in all four directions;
- *Direct Lines*: route signals to neighboring blocks – vertically, horizontally, and diagonally;
- *Fast Lines*: internal CLB local interconnections from LUT outputs to LUT inputs.

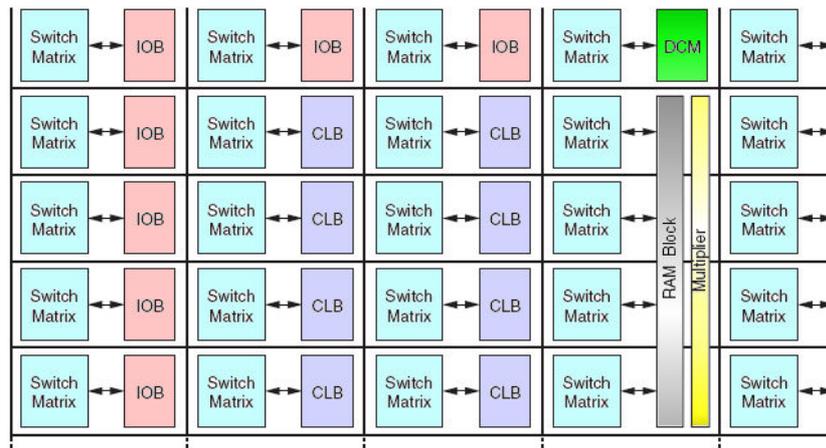


Figure A.3.7: Routing in Virtex II device. [29]

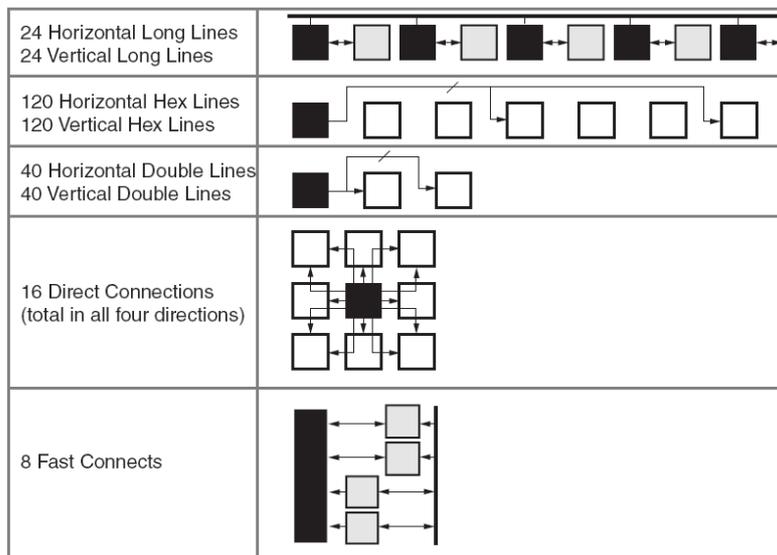


Figure A.3.8: Hierarchical routing resources in Virtex-II. [29]

Appendix B

ARCHITECTURE OF TIGERSHARC ADSP-TS201 PROCESSOR

The *TigerSHARC ADSP-TS201* processor (Figure B.1.1) is a static superscalar 128-bit processor of DSP type suited for fixed and floating point operations. Fixed and floating point operations are performed with the same amount of processor core clock cycles. The processor operates at 600 MHz clock frequency, and consists of the following main parts: two computational blocks, 24M bit of on-chip Dynamic RAM (DRAM) with six 4K word caches (one per memory block), integrated I/O peripherals, a host processor interface, Direct Memory Access (DMA) controllers, Low-Voltage Differential Signaling (LVDS) link ports, and shared bus connectivity for multiprocessing. The main features of the ADSP-TS201 processor are: [\[32\]](#)

- Dual computational blocks: X and Y – each consisting of a *Arithmetic Logic Unit (ALU)*, multiplier, *Communications Logic Unit (CLU)*, shifter and a 32-word register file;
- Dual *Integer ALUs (IALUs)* J and K , each contains a 32-word register file;
- Program sequencer – controls the program flow and contains an *Instruction Alignment Buffer (IAB)*, *Branch Target Buffer (BTB)*, *Program Counter (PC)*, address fetch mechanism and interrupt manager;
- Three 128-bit buses providing high bandwidth connectivity between internal memory and the rest of the processor core (computational blocks, IALUs, program sequencer, and System-On-a-Chip (SOC) interface);
- A 128-bit bus providing high bandwidth connectivity between internal memory and external I/O peripherals (DMA, external port, and link ports);
- 24M bits of internal memory organized as six 4M bit blocks. Each block contains 128K words by 32 bits and connects to the crossbar through its own buffers and a 128K bit four way set associative cache.

B.1. Computational Blocks

As illustrated in Figure B.1.1, the ADSP-TS201 processor core includes the two computational blocks X and Y. Each of these blocks contains a 32 by 32-bit register file and four independent computation units: ALU, multiplier, CLU and shifter. The computational blocks and their units can operate in parallel.

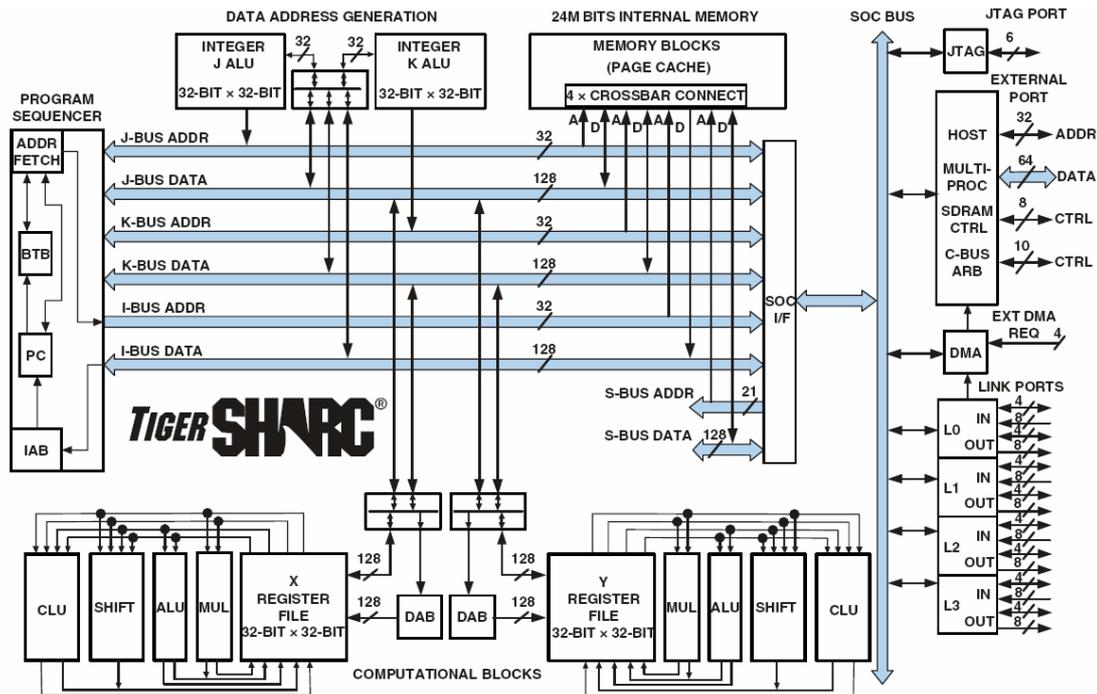


Figure B.1.1: Architecture of the ADSP-TS201 processor core. [33]

B.1.1. Arithmetic Logic Unit (ALU)

ALU, shown in Figure B.1.1, is a 64-bit unit. It performs arithmetic operations (addition, subtraction) on fixed point and floating point data. Moreover, it performs logical and data conversion (expand, compact) operations on fixed point data. The source and destination of most ALU operations is the register file. The register file in computational block X consists of 32 registers (XR0 through XR31), and the register file in Y consists of YR0 through YR31 registers. Depending on data type (fixed or floating point) ALU can support parallel operation on different length (8, 16, 32, 64 bit) operands. [34]

Figure B.1.2 illustrates an example of the parallel addition on fixed point, byte size data.

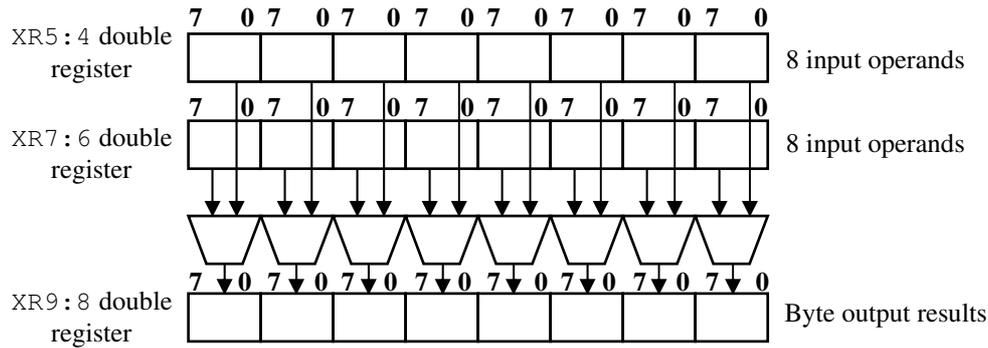


Figure B.1.2: An example of parallel addition on fixed point, byte size data.

B.1.2. Multiplier

A multiplier unit (Figure B.1.1) performs multiply operations on both fixed and floating point data, and performs multiply-accumulate (MAC) operations on fixed point data. The multiplier takes operands from the register file, and returns the result to the register file or to the one of the special purpose registers (e.g., accumulator). The TigerSHARC ADSP-TS201 processor uses the MR accumulator (Figure B.1.3) to store the result of fixed point MAC operations. The multiplier transfers the result of the MR register to the register file before the other accumulate operation are proceeded. A multiplier unit also performs complex number MAC operations on fixed point data, and executes data compaction operations on accumulated results when moving data to the register file in fixed point formats. [34]

The example of fixed point 32-bit MAC is shown in Figure B.1.3.

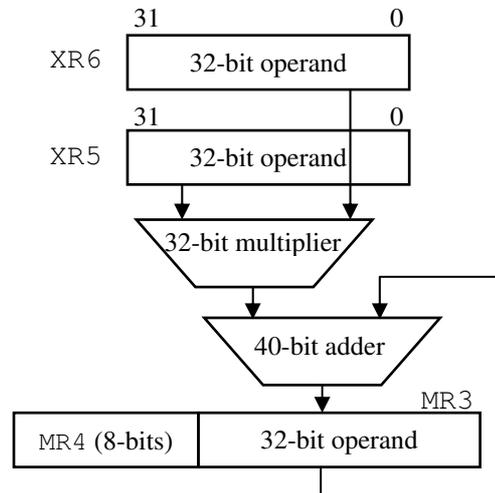


Figure B.1.3: Fixed point 32-bit multiply-accumulate (MAC) operation.

In Figure B.1.3, we can see that the result of MAC operation is 32-bit wide. The additional M4 register's eight bits store the overflow of MAC operations.

B.1.3. Shifter

The shifter is a 64-bit unit, which can operate on one 64-bit, one or two 32-bit, two or four 16-bit, four or eight 8-bit fixed point operands. The shifter takes operands from the register file and returns the result to the register file as well. A shifting unit performs the following operations: [34]

- Shift and rotate bit field, from off-scale left to off-scale right;
- Bit manipulation (bit set, clear, toggle, and test);
- Bit field manipulation (field extract and deposit);
- Scaling factor identification, 16-bit block floating-point;
- Extract exponent;
- Count number of leading ones or zeros.

The logical shift and arithmetic shift operations are shown as examples in Figures B.1.4 and B.1.5. The operand stored in R5 register is shifted by four (the number stored in R4 register) and the result is stored in R6 register. The left shift is proceeded when the value in R4 register is positive; and to the right when it is negative.

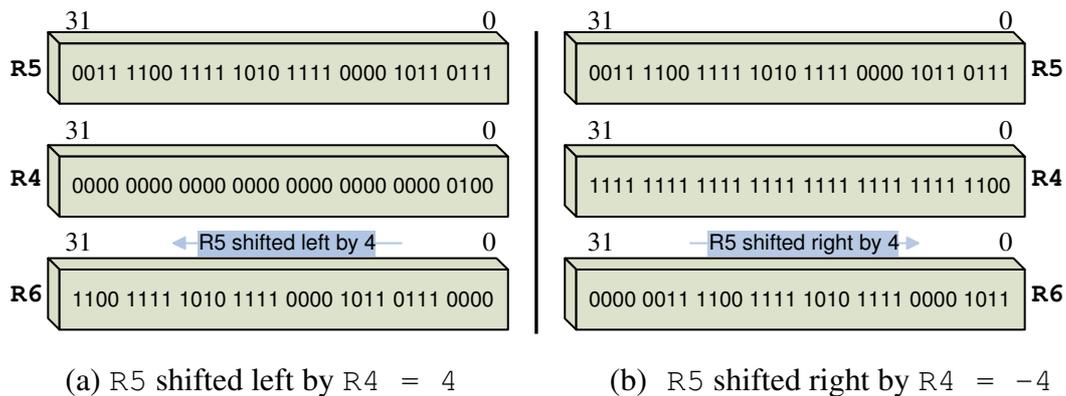


Figure B.1.4: Logical shift by four: (a) shift to the left, (b) shift to the right.

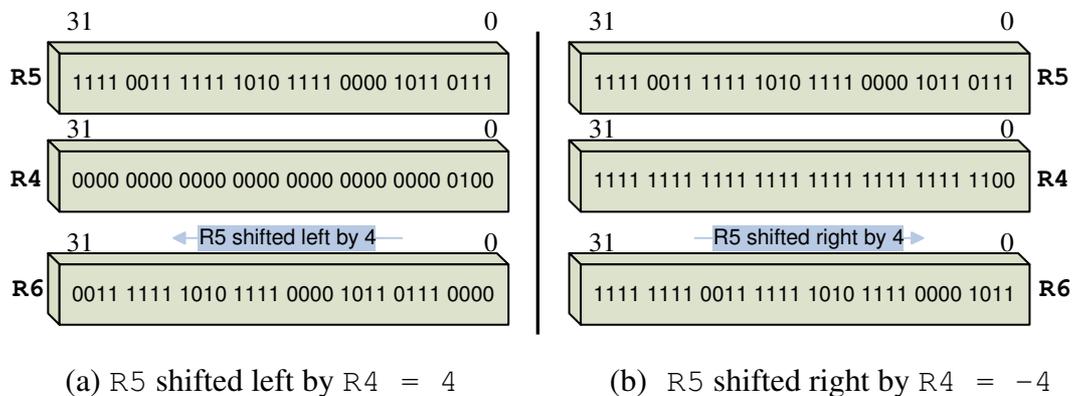


Figure B.1.5: Arithmetic shift by four: (a) shift to the left, (b) shift to the right.

B.1.4. Communications Logic Unit (CLU)

“The communicational logic unit (CLU) is a 128-bit unit, which performs specialized communications functions. It supports Viterbi decoding, turbo-code decoding, Code Division Multiple access (CDMA) decoding, despreading operations, and complex correlation. The functions also can be applied in non-communicational algorithms. CLU includes 32 Trellis Registers (TR) and 4 Trellis History Register (THR). CLU takes its inputs from the register file or TR and THR registers, and then returns its result to the register file or TR and THR registers. CLU operates on fixed point data” [34]. A CLU unit supports the following operations:

- Jacobian logarithm for turbo decode ($TMAX$);
- CDMA despreader ($DESPREAD$);
- CDMA cross correlations ($XCORRS$);
- Polynomial reordering ($PERMUTE$);
- Trellis add, compare, select (ACS).

B.2. Integer ALU

The ADSP-TS201 processor core contains the two independent integer ALUs (IALUs), denoted as J-IALU and K-IALU (Figure B.1.1). IALUs support regular fixed point ALU operations and data addressing operations. IALUs provide memory addresses when data is transferred between memory and registers. IALUs enable simultaneous addresses for two memory accesses (read or write). IALUs contain two register files ($J0$ through $J31$ registers for J-IALU, and $K0$ through $K31$ registers for K-IALU) and eight dedicated registers for circular buffer addressing. All IALU registers are 32-bit wide, memory-mapped, universal registers. [34]

IALU data addressing operations provide memory read and write access for loading data to registers and storing to memory. For memory reads and writes, the IALU provides two types of addressing: direct and indirect.

B.3. Program Sequencer

The ADSP-TS201 processor core also contains a program sequencer (Figure B.1.1) for managing program execution. The main sequencer functions are following: [34]

- Instruction fetch from memory;
- Instruction line extraction from the fetched data;
- Instruction line decoding for sending the instruction to particular execution unit;

- Program flow control instruction execution;
- Program stalls monitoring.

The mentioned functions are performed by the following program sequencer main elements: instruction alignment buffer (IAB) and branch target buffer (BTB).

B.3.1. Instruction Line Structure

The ADSP-TS201 processor executes from one to four 32-bit *instruction slots* in an instruction line. An instruction is a 32-bit word that activates one or more of the TigerSHARC processor's execution units to carry out an operation. Instruction line (Figure B.3.1) consists of up to four 32-bit instruction slots. Instructions on an instruction line are executed in parallel. One instruction line is executed with a throughput of one processor core clock cycle. [34]

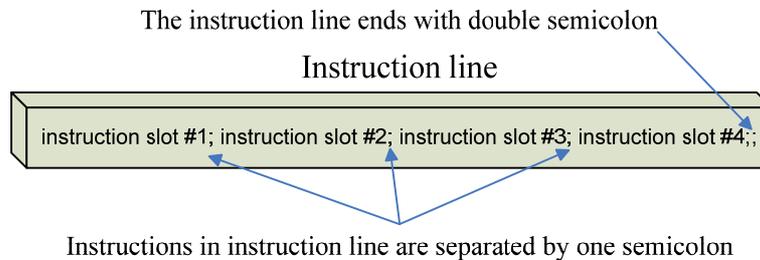


Figure B.3.1: Instruction line structure.

B.3.2. Instruction Alignment Buffer (IAB)

Instruction alignment buffer (Figure B.3.2) is a six quad word FIFO buffer. It buffers the fetched instructions and keeps the fetch unit independent from the rest of the instruction pipeline. This independence lets the fetch unit to run when the other parts of the pipeline are stalled. IAB also aligns the fetched words to instruction lines and distributes for execution. The alignment guarantees that instruction lines are able to execute in parallel. The sequencer fetches a quad-word wide instruction from memory and then writes the word into the IAB. Program sequencer extracts instruction lines consisting of one, two, three or four instructions from IAB for the processor to decode or execute. [34]

The Most Significant Bit (MSB) of an instruction word (Figure B.3.2) indicates whether it is a regular instruction slot or the slot at the end of an instruction line. If the MSB is equal to 0, it means that it is a regular instruction slot, which ends with single semicolon (Figure B.3.1). MSB equal to 1 indicates the end of instruction line, which ends with double semicolon (Figure B.3.1).

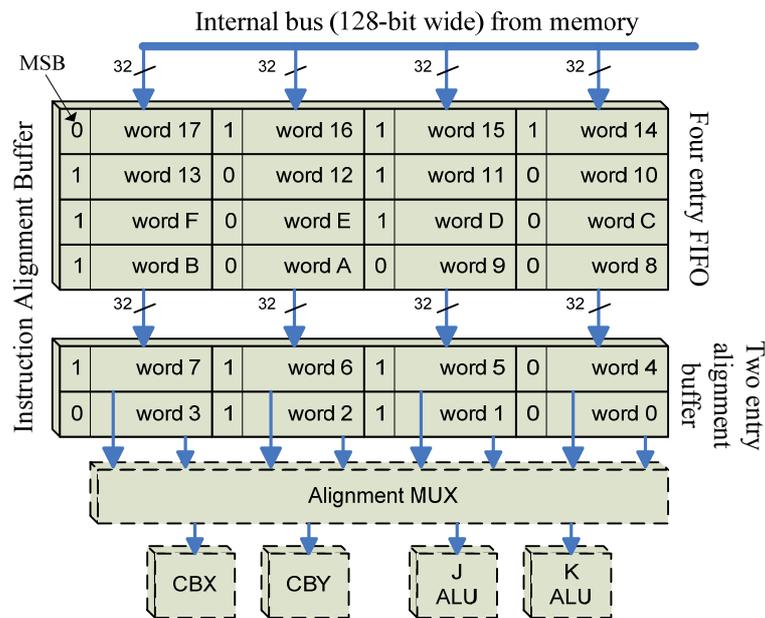


Figure B.3.2: Structure of Instruction Alignment Buffer.

B.3.3. Branch Target Buffer (BTB)

“Other significant program sequencer part is branch target buffer (BTB). It is used to reduce branch delays for conditional and unconditional instructions. The program sequencer writes information about every predicted branch into BTB. The BTB entries make up an associative memory that records a history of predicted branches. The BTB examines the flow of addresses during the first pipeline stage. When the BTB recognizes the address of an instruction that caused a jump on a previous pass of the program code it substitutes the corresponding destination address as the fetch address for the following instruction. When a branch is currently cached and correctly predicted, the performance loss due to branching is reduced from either nine or five stall cycles to zero.” [34]

B.4. Memory and Buses

Some of microprocessors use a single address and single data bus for memory access. This type of memory architecture is called *Von Neumann* architecture. However, DSP processors may require greater data throughput than Von Neumann architecture provides, and so many DSPs use memory architectures that have separate address and data buses for program and data storage. The two sets of buses let the processor fetch a data word and an instruction simultaneously. This type of memory architecture is called *Harvard* architecture. TigerSHARC DSP uses a *Super Harvard* architecture. This memory architecture has program and data buses, but provides a single, unified address space for program and data storage. The data memory bus carries only data, and the program memory bus handles instructions or data.

B.4.1. Buses

The ADSP-TS201 processor architecture contains four buses (J-bus, K-bus, I-bus, S-bus) connected to its internal memory (Figure B.4.1). J-bus, K-bus and I-bus are used for memory accesses to computation units, integer ALUs and program sequencer (Figure B.1.1). S-bus is used for memory accesses to processor's peripherals (external port and link ports). S-bus also can provide data transfers between internal memory and other processors without interrupting main processor's core accesses to its memory. During a single cycle, the processor core uses the independent J-bus and K-bus for simultaneous access to data from two different memory blocks. The processor's internal bus architecture lets the core and I/O access twelve 32-bit data words and four 32-bit instructions each cycle.

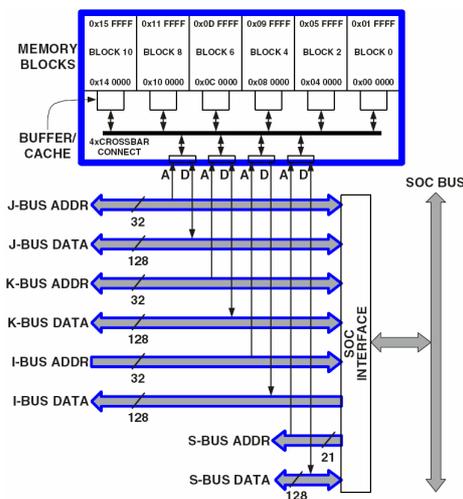


Figure B.4.1: Structure of processor's internal memory and buses. [34]

B.4.2. Memory

The ADSP-TS201 processor core contains a large embedded DRAM internal memory and provides access to external memory through the processor's external port. The processor's internal memory has 24M bits of embedded DRAM memory. The memory is divided into six blocks of 4M bits (128K words of 32 bits). Instructions and data can be stored in each memory block. When the data and instructions are placed in different memory blocks, the processor is able to access data and fetch instructions simultaneously, because one access to particular memory blocks is available per processor clock cycle. Each memory block contains a 128K bit cache, which enables accesses with no latency induced by bus stalls. Internal memory blocks connect to the four 128-bit wide internal buses through a crossbar interface connection (Figure B.4.1), enabling the processor to perform four memory transfers in the same cycle. The embedded DRAM works in a one half frequency clock. The largest access to the embedded DRAM is of 8 words, 256 bits. To keep full execution performance, the memory system provides the processor core with up to 4 accesses per cycle from different blocks. [34]

Bibliography

- [1] *Asymmetric Digital Subscriber Line (ADSL) Transceivers*. ITU Recommendation G.992.1, 1999.
- [2] John A. C. Bingham. *ADSL, VDSL, and Multicarrier Modulation*. John Wiley & Sons Ltd, 2000.
- [3] Charles K. Summers. *ADSL: Standards, Implementation, and Architecture*. CRC Press, 1999.
- [4] *Cyclic Redundancy Check*. Available from: <http://www.relisoft.com/Science/CrcMath.html>
- [5] R. L. Brewster. *Data Communications and Networks 3*. Institution of Electrical Engineers (IEE), 1994.
- [6] J. Cioffi. *A Multicarrier Primer*. Amati Communications Corporation and Stanford University, 1991.
- [7] Peter Sweeney. *Error Control Coding from Theory to Practice*. John Wiley & Sons Ltd, 2002.
- [8] S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice Hall: Englewood Cliffs, NJ, 1983.
- [9] J. G. Moreira and P.G. Farrell. *Essentials of Error Control Coding*. John Wiley & Sons Ltd, 2006.
- [10] Todd K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. John Wiley & Sons Ltd, 2005.
- [11] B. Sklar. *Reed-Solomon Codes*, April 2002. Available from: <http://www.phptr.com/articles/article.asp?p=26335&rl=1>
- [12] *Primer: Reed-Solomon Error Correction Codes (ECC)*. Comtech AHA Corporation, 1995.
- [13] Qian Hongyi. *On the Performance of Reed-Solomon, Convolutional and Parity Check Codes for BWA Applications*. IEEE 802.16.1pc-00/37, 2000.
- [14] Charan Langton. *All About Modulation - Part I*. Available from: <http://www.complextoreal.com/tutorial.htm>

- [15] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto. *Affinity-Driven System Design Exploration for Heterogeneous Multiprocessor SoC*. IEEE Trans. Computers (Vol. 55, No. 5): pp. 508-519, May 2006.
- [16] T. Cooper. *Taming the SHARC*. Technical report, Ixthos Inc., 2000.
- [17] Yannick Le Moullec, N. Ben Amor, J-Ph. Diguët, M. Abid, J-L. Philippe. *Multi-Granularity Metrics for the Era of Strongly Personalized SOCs*. Universite de Bretagne Sud, Lorient, France, 2ENIS engineering school, Sfax, Tunisia, 2003.
- [18] A. Saramentovas, P. Ruzgys. *Guidance for the Implementation of HSDPA by Means of Design Space Exploration with Design-Trotter*. Aalborg University, 2007.
- [19] Yannick Le Moullec, Søren S. Christensen, Wen Chenpeng, Peter Koch, Sebastien Bilavarn. *Fast System-Level Design of Wireless Applications*. WPMC05, September 2005, Aalborg, Denmark.
- [20] Yannick Le Moullec, Jean-Philippe Diguët, Dominique Heller, Jean-Luc Philippe. *Fast and Adaptive Data-flow and Data-transfer Scheduling for Large Design Space Exploration*. ACM/SIGDA GLSVLSI02, April 2002, New-York, USA.
- [21] Jean-Pierre Deschamps, Gery Jean Antoine Bioul, Gustavo D. Sutter. *Synthesis of Arithmetic Circuits – FPGA, ASIC and Embedded Systems*. John Wiley & Sons, Inc., 2006.
- [22] *Xilinx ISE Overview*:
<http://toolbox.xilinx.com/docsan/xilinx8/help/iseguide/iseguide.htm>
- [23] Celoxica. *Handel-C language reference manual (for DK version 4)*. Product manual, 2005.
- [24] *Handel-C Code Optimization*. Celoxica Limited, 2003.
- [25] *Introduction to Assembly Language*. Available from:
<http://www.osdata.com/topic/language/asm/asmintro.htm>
- [26] Analog Devices, Inc. *VisualDSP++ 4.0 Getting Started Guide (Revision 1.0)*, January 2005.
- [27] Analog Devices, Inc. *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for TigerSHARC Processors (Revision 2.0)*, January 2005.
- [28] *Avnet distributor*: <http://www.avnet.com/>

- [29] Jean-Pierre Deschamps, Gery Jean Antoine Bioul, Gustavo D. Sutter. *Synthesis of Arithmetic Circuits – FPGA, ASIC and Embedded Systems*. John Wiley & Sons, Inc., 2006.
- [30] Clive “Max” Maxfield. *The Design Warrior’s Guide to FPGAs*. Mentor Graphics Corporation and Xilinx, Inc., 2004.
- [31] Xilinx, Inc., *Virtex-II Platform FPGAs: Complete Data Sheet*, DS031 (v3.4), March 1, 2005. Available from: <http://www.xilinx.com>
- [32] Analog Devices Inc. *ADSP-TS201 TigerSHARC Processor Hardware Reference* (Revision 1.0), November 2004.
- [33] Analog Devices Inc. *ADSP-TS201 TigerSHARC Embedded Processor Data Sheet* (Rev. C), 2006.
- [34] Analog Devices Inc. *ADSP-TS201 TigerSHARC Processor Programming Reference* (Revision 1.1), April 2005.