Rapid System Prototyping for Error-Control Coding in Optical CDMA Networks

Martin Irman

Department of Electrical and Computer Engineering
McGill University
Montreal, Canada

August 2005

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

Your file Votre référence
ISBN: 978-0-494-24969-7
Our file Notre référence
ISBN: 978-0-494-24969-7

Abstract

With increasing bandwidth requirements of individual users, *fibre-to-the-home* systems are promising candidates for *last mile* communication. In case of local area networks, optical CDMA with nonorthogonal spreading sequences has emerged as an attractive technology that can manage quickly varying user requirements, while enabling *total bandwidth utilization*, avoiding *network congestion* and preventing *denial of service*. However, powerful *error-control codes* have to be utilized within optical CDMA systems to prevent errors caused by multiuser interference. Due to data rates common in optical communication, dedicated hardware is required to run such *error-control codes* at very high speeds, e.g., 155 Mbps or 652 Mbps.

This work presents a rapid system prototyping platform for *error-control codes* which are to be incorporated into the above mentioned optical CDMA systems. The platform consists of a design methodology, an extensive library of modules and an environment for testing designed specifically for optical CDMA systems but applicable to other communication systems as well. It is built on *System Generator* from *Xilinx*, a *Matlab/Simulink* based visual design tool, and enables a "push of a button" transition from code specification to real-time implementation on an FPGA chip.

Initially, both the hardware specifics of this platform and the details of the developed modular design methodology are presented. Consequently, implementation of a custom design construct (a *Generate* block) and a library of communication system modules are described together with blocks and methodology for testing the developed algorithms. Finally, design and evaluation of three different communication systems are presented. These designs show that the platform can be used for prototyping and evaluation of error-control algorithms running at the required processing speeds mentioned above.

Sommaire

En réponse à une demande croissante de vitesse de chargement pour les individus, les systèmes « fibre optique jusqu'au domicile » sont de candidats intéressants pour les communications de dernier kilomètre. Dans le cas de réseaux locaux, l'accès multiple par répartition en code (AMRC) optique avec séquences de codes non orthogonales s'est démontré être une solution intéressante, qui entre outre s'adapte au va et viens des utilisateurs tout en maintenant l'utilisation totale de la bande passante et en évitant la congestion du réseau et le refus de service.

Cependant, des codes de correction d'erreur puissants doivent faire partie du système d'AMRC optique afin d'en éliminer les erreurs dues à l'interférence entre les utilisateurs. Comme ces codes présentent de très hauts débits, le matériel qui leur est dédié devra fonctionner à très grande vitesse, par exemple 155 Mbps ou 652 Mbps.

Cette dissertation présente une plateforme de développement de prototype rapide pour les codes de correction d'erreurs incorporés dans les systèmes AMRC décrits ci-haut. Cette plateforme consiste d'une méthodologie de conception, une librairie de modules extensive et un environnement permettant la validation des prototypes conçu spécifiquement pour les systèmes d'AMRC optiques mais applicable à d'autres systèmes de communication aussi bien. Cette plateforme est créée à l'aide de System Generator de Xilinx, un outil basé sur Matlab/Simulink, et permet la transformation automatique de spécifications codées en implémentation temps réel sur un réseau prédiffusé programmable par l'utilisateur. En premier lieu, nous présentons les détails du matériel spécifique à cette plateforme et la méthodologie de conception que nous lui associons. Ensuite, l'implémentation d'un bloc de génération automatique et une librairie de modules de communication sont décrites, ainsi que les blocs et la méthodologie utilisés pour la validation des algorithmes.

Finalement, la conception et l'évaluation de trois systèmes de communication sont présentées. Nous démontrons que cette plateforme peut être utilisée pour créer des prototypes et évaluer des algorithmes de correction d'erreurs aux vitesses mentionnées ci-haut.

Acknowledgements

I would like to express my gratitude to my advisor, Professor Jan Bajcsy, for his patience, valuable insights and extensive feedback. His support and guidance made my stay here at McGill University pleasant and enriching. Working on hardware design and coding has been a truly great experience and I am thankful to Professor Jan Bajcsy that he exposed me to this magnificent field.

My special thanks go to my colleagues and friends who helped me in forming the ideas presented in this thesis. Naveen Mysore, Aminata Amadou Garba, Phillip Sawbridge, Isabel Deslauriers, Xu Bo, Samer Lutfi and Mehmet Akçakaya helped create and intellectual environment that always pushed me forward.

Additionally, I would like to acknowledge McGill University, Natural Sciences and Engineering Research Council of Canada, Le Fonds Québécois de la Recherche sur la Nature at les Technologies, Canada Foundation for Innovation and Xilinx Corp. for support during various stages of preparation of this thesis.

Above all, I would like to thank my girlfriend Eva Wang, my father František, my mother Eva and my brother Vladimír for continuous support and encouragement. Without the gentle pressure and encouragement of my girlfriend, this work would be much harder to write. It is to her and my family that I dedicate this thesis.

It is a mistake to think you can solve any major problems just with potatoes.

– Douglas Adams

Contents

List of Figures

List of Tables

List of Acronyms

ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BSC	Binary Symmetric Channel
CDMA	Code Domain Multiple Access
CLB	Configurable Logic Block
DSP	Digital Signal Processing
FEC	Forward Error Correction
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
IOB	Input/Output Block
LUT	Look-Up Table
LVDS	Low-Voltage Differential Signalling
OCDMA	Optical Code Domain Multiple Access
SINR	Signal to Interference and Noise Ratio
SNR	Signal to Noise Ratio

List of Applied Terms

symbol	An entity represented by the state of a binary bus with a specific number of bits.
packet	A sequence of *symbols* transmitted over a signal bus consisting of a *packet header* followed by the *packet data* .
packet header	First three *symbols* of a *packet* specifying packet parameters.
packet type	Numeric user defined value included in the *packet header.*
packet data	A sequence of *symbols* of length given by the *packet header.*
block	*Simulink* subsystem implementing specific functionality.
module	A *block* processing data organized in packets.
line	*Simulink* signal interconnect used to transmit packets
schematic	*Simulink* diagram consisting of interconnected *blocks*
port	An *input port* or an *output port.*
input port	A block making a *signal*, with a specific label in a specific context, available.
output port	A block assigning a label to a signal in a specific context.
nesting	Use of a concept in a design based on an identical concept.
signal width	The number of bits of a binary representation of a *block's* input/output.
signal binary point	The position of a binary point in the representation of a signal enabling interpreting the signal as a real number.
subsystem	*Block* representing a *Simulink schematics* and defining inputs and outputs of this *schematics.*
mask dialog	A *Simulink* dialog box associated with a *block* which enables the user to enter *block* specific parameters.
mask parameter	A parameter within the *mask dialog* of a *block.*

Chapter 1

Introduction

1.1 Motivation

Optical systems are established in long haul data transmission where these offer the required transmission capacity [1]. With increasing bandwidth requirements of individual end-users, fibre based systems are a promising candidate for the *last mile* communication. In the case of local or regional networks, optical CDMA with nonorthogonal users' signatures, as described in [27] [29] and [30], is emerging as a network protocol that can manage the quickly varying user requirements while enabling *total bandwidth utilization*, avoiding *congestion* and preventing *denial of service*.

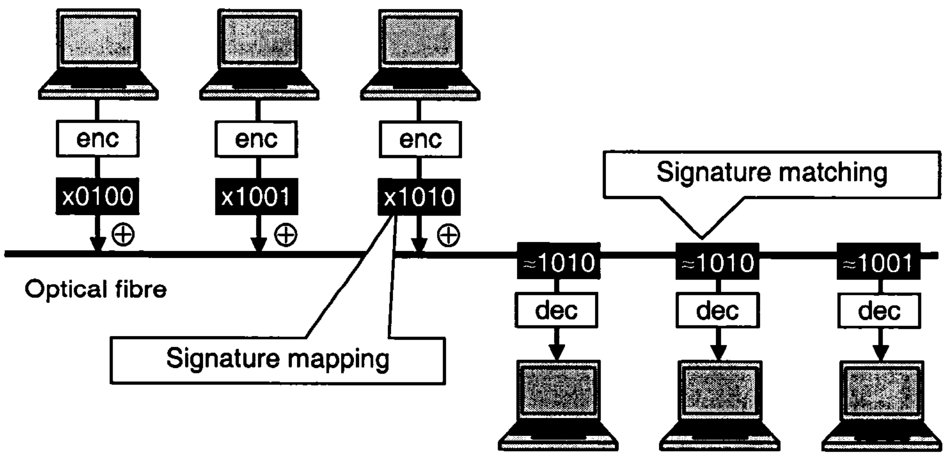


Figure 1.1: Scheme of an optical CDMA network

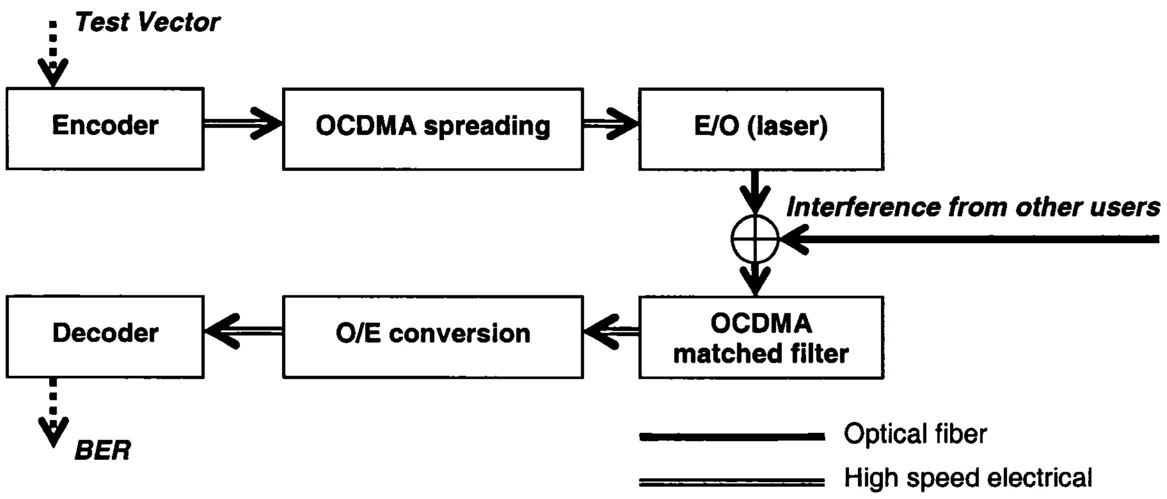


Figure 1.2: Optical CDMA prototype as a point to point communication system

In an optical CDMA network, the data from multiple users is merged onto an optical fibre as depicted in Figure 1.1. Therefore, every transmission on the network will suffer from the interference caused by other users and powerful error-control codes have to be utilized to correct these errors [9] [11]. Only with recent advances in electronics, it is possible to employ powerful computationally intensive codes (e.g., *Turbo Codes* [4]) at the data rates seen in optical systems while utilizing inexpensive hardware.

This work presents a rapid system prototyping platform for error-control codes which are to be used in an optical CDMA network prototype. The prototype will be used to test the performance of the system while implementing the functionality of a single user and emulating the interference caused by other users. This prototype will be modular and its components are depicted in Figure 1.2. In this work, we are concerned only with the *encoder* and the *decoder* modules which will be later integrated into the prototype. We present a platform based on *System Generator* from *Xilinx* that enables fast design, verification and performance testing of error-control codes for these modules.

The platform was implemented specifically for prototyping optical CDMA systems but many components of the platform are generic and can be used for development of other communication systems (e.g., wireless or wireline). The developed *Generate* block can be a utilized to simplify any structured hardware design and the modular design methodology, many of the implemented modules and the performance testing tools can be applied to communication systems in general.

The goal of this thesis is to use the platform to implement error-control at speeds above 155 Mbps. The results in this thesis were in part presented in [17], [18] and the platform was demonstrated at [11]. The specifics of the interconnection between the user, the *encoder/decoder* hardware and the optical hardware as well as the specifics of the optical CDMA transmission and the optical matched filter are beyond the scope of this thesis and are briefly addressed in the conclusion. In our setup, the *encoder* and the *decoder* have been connected directly and the noise was added using a pseudo-random noise generator run in hardware.

1.2 Implementing a Flexible Design Platform

The high speed of optical communication is pushing real-time implementations of signal processing algorithms for these systems from software to hardware platforms. The latter technology allows key soft-decision decoding algorithms (e.g., Viterbi [36] [10], SOVA [15], BCJR [3] [14]) to posses a throughput rate of hundreds of megabits per second. Therefore, to achieve the required speed, our platform will be based on programmable hardware technology. We present an implementation of a rapid system prototyping platform that enables the development of fully customizable communication systems for use in optical CDMA transmission. Full customizability is important for research purposes as well as for quick deployment of re-configurable systems.

While there is a fairly standard suite of *software development* tools allowing a "push of a button" transition from customizable specification (source-code) to an implementation (running application), the world of *hardware development* tools is much more complex. Various design paths are used in practical hardware development to proceed from the specification to a hardware (FPGA or ASIC) prototype. In many instances, this process consists of a number of steps, where the output of one software tool is the input to another tool.

In case of *software development*, the design process usually consists of numerous steps hidden from the user with intermediate results in an internal format that is not human-readable, i.e., suitable only for processing by the appropriate tool. On the other hand, in *hardware development*, depending on the design path, the intermediate results are in a well-defined human-readable language (e.g., *VHDL* [2]) and are expected to be modified by the developer. This breaks the chain of a "push of a button" transition from specification to a functional prototype, because modifications of intermediate results cannot be incorporated into the original specifications. Moreover, it is hard to track what modifications have been made and, once the necessity arises to regenerate the intermediate results, it might be hard to apply these modifications. Therefore, it is highly desirable to design a hardware development environment that would allow the design of customizable algorithms with a "push of a button" transition from specifications to implementation.

Hardware prototyping is usually done using a programmable logic chip (e.g., an FPGA). The advantage of developing a design on an FPGA is easy re-programmability which enables direct verification of the prototype under development. Another advantage of an FPGA is that using this chip in production units shortens the time-to-market and lowers the fixed production cost as there is no necessity of manufacturing of an Application-Specific Integrated Circuit (ASIC). The low fixed cost makes an FPGA suitable for use in units manufactured in small quantities, such as telecommunications systems switches and base-stations. For high quantity consumer electronics, such as cell phones, designing an ASIC might be more cost effective. Nonetheless, prototyping of such devices is done using an FPGA, where an ASIC can be designed and manufactured as a direct copy of the logic implemented on the programmable logic chip.

The platform developed in this work facilitates seamless transition from an error correcting code specification to the bit stream that can be uploaded directly onto an FPGA chip. We describe the design methodology employed in the prototyping platform, tools for testing design, the testbed used in verifying the design and custom enhancements of the design environment. We have developed an environment that effectively allows the user to modify most of the parameters of an encoding system from a top-level module view and the system automatically propagates these changes to the low level building components. This ease of customizability and the high throughput of the systems implemented in hardware will allow us to execute reliability tests in scenarios where the error rate is below 10^{-9} within several hours. In contrast, conventional simulation in *C/C++* or *Matlab* could require weeks to finish. This will allow us to compare scenarios by considering different factors, such as: the error correcting capabilities of the decoder, size of the design and speed in real time operation.

1.3 Project Description

In the setup phase of our project, to achieve the above stipulated goal of easy implementation and testing of highly customizable algorithms, we have:

- Selected the platform for hardware co-simulation (*Nallatech Xtreme DSP Kit* development board with a Virtex II FPGA chip) and set-up the software environment necessary to communicate with and compile for this platform.
- Chosen *System Generator* from *Xilinx* [43] to implement our design. This tool allows us to program an FPGA by defining our implementation using basic hardware elements in a visual environment based on *Matlab/Simulink* [20]. The tool enables us to clearly define modules and their interfaces. Furthermore, it uses *Matlab* as a powerful scripting language which allows us to dynamically create custom algorithmic constructs.

In the following stage, we have developed a design platform that consists of software environment, methodologies and architectures to facilitate fast and easy design, testing and verification of digital communication algorithms. Specifically, in this process of developing a hardware design platform, we have:

- Created a library of modules that perform signal processing tasks. These blocks can be used as building blocks of different algorithms in communications.

- Defined a system for dispatch and scheduling of packets that allows identifying the processing bottlenecks and parallelizing the necessary components to achieve higher data throughput.
- Developed a special "Generate" block that allows defining customizable regular structures within the *System Generator* environment. The *Generate* block is used inside many modules in the library mentioned in the first bullet to achieve customization where parallel processing is necessary for effective implementation.
- Established an environment that allows automated execution of bit error rate simulations (i.e., an automated testbed) with different channel characteristics on an FPGA. This is a system that uses *System Generator* capabilities to control simulations on an FPGA from the host PC and stores the simulation results in *Matlab.*

Finally, we have used the developed hardware design platform in the design of several communication algorithms. In particular, we have:

- Implemented a *Convolutional Codes Encoder* with a *Viterbi Decoder*, *Block Codes Encoder* with a hard decision *Syndrome Decoder* to test and demonstrate the functionality of our development platform at high speeds.
- Designed a soft decision *Turbo Product Codes (TPC) Decoder*, which allows the implementation of communication system using an average sized FPGA with throughputs over 200 Mbps over *additive white Gaussian noise channel* (AWGN).
- Verified the bit-error-rate performance of these coding schemes using the design platform's automated testbed.

1.4 Organization of this Thesis

In this thesis, we present a development platform for rapid prototyping and hardware co-simulation of coded communication systems. Following this introductory chapter, Chapters 2 through 6 present the work on this platform and the implementation of an error control scheme for optical CDMA. Specifically:

- Chapter 2 introduces the initial hardware and software setup and elaborates on design options for hardware development.
- Chapter 3 describes the modular design methodology employed in this project. The chapter outlines the rules for data exchange between modules in the form of packets, proposes standards for the module customization user interface and describes the implementation of modules that conform to this specification. This methodology was in part presented in [18].
- Chapter 4 focuses on the software implementation of an enhancement on *System Generator* that enables automatic generation of arrays of regular structures in this specification language. This tool, which we call "the *Generate* block", enables the design of generic modules with parameters specifiable in the front end. The implementation of this tool was in part presented in [17].
- Chapter 5 explains the usage of the platform for rapid prototyping. It describes a library of communication system building blocks (based on previous chapters), tools for debugging and a testbed for automated performance evaluation.
- Chapter 6 focuses on implemented solutions. It presents the implementation of several error-control schemes and evaluates their performance by simulations. The chapter further discusses the advantages of the implemented *Turbo Product Code* over other *Turbo Coded* schemes and, finally, compares the implemented code to similar commercially available solutions.
- Chapter 7 concludes with a summary of our work and an outline for future development of the platform into a standalone prototype.

Appended to this work are three technical sections:

- Appendix A presents a step-by-step workstation installation guide.
- Appendix B provides the developer with documentation of the modular library.
- Appendix C includes the platform implementation on an attached compact disk.

Chapter 2

Initial Project Setup

In this chapter, we outline the options considered and decisions made that formed our initial software and hardware system setup. First, we give an introduction into the various design options for high performance communication systems, specifically hardware design using *System Generator* from *Xilinx*. Next, we list error-control system implementations available commercially for comparison with our system. Finally, we introduce our design platform based on *System Generator* and a development board from *Nallatech*.

The reader may want to skip the introduction into programmable logic hardware, HDL or *System Generator* design if he/she is familiar with these subjects. Nevertheless, Section 2.4 contains details on the setup of our specific development platform.

2.1 Design Path Options in Software and Hardware Development

When considering the implementation of forward error correction with high throughput we have several options of what hardware to use. We can:

- Use a common high performance processor (such as an Intel Pentium 4),
- Use a processor optimized for Digital Signal Processing (DSP processor),
- Use a programmable hardware chip (such as an FPGA),
- Create an Application-Specific Integrated Circuit (ASIC).

Table 2.1: Performance and processing requirements comparison

Unit / Algorithm	Performance / Requirement
Intel Pentium 4, 3.6Ghz	9000 MIPS
DSP Processor	1200 – 8000 MIPS
FPGA (XC2V2000)	60 000 MIPS, roughly compares to
ASIC	approx. 20% better than a comparable FPGA
Syndrome (7,4) Block Decoder at 155 Mbps	≈190 MIPS
Turbo Decoding at 1 Mbps (TCC)	≈1000 MIPS
Turbo Decoding at 155 Mbps (TCC)	≈150 000 MIPS

The design process substantially differs for the first two and the second two options. In the first two cases, where an instruction based processor is used, a sequential language such as *C/C++* or *Matlab* can be used to implement our design. The simple data model and the powerful data manipulation and computational techniques of these languages enable us to implement algorithms (software design) that are easily customizable both at design time as well as run time. On the other hand, describing the behaviour of a hardware implementation (hardware design) on an FPGA or an ASIC is much more complicated.

2.1.1 Performance of Instruction Based Processors

Solutions based on processor architectures are not able to achieve performance sufficient to decode complex codes (e.g., Turbo Codes) at high speeds. Processor performance can be measured in Million Instructions Per Second (MIPS). In reference [13], a low-complexity Turbo Decoder is presented which enables decoding at an average throughput of 70kbit/s on a 40 MIPS DSP processor. More recently, a high-speed implementation of Turbo Codes is available in reference [8]. This implementation achieves the throughput of approximately 3 Mbit/s on a 4500 MIPS desktop processor. These results justify a rough estimate of 1000 instructions to be the complexity of decoding a single Turbo Coded bit.

The latest desktop processors have a performance just below 10000 MIPS. Specifically, the Intel Pentium 4 running at 3.6 GHz is rated at 9000 MIPS [16].

The performance of DSP processors is similar. State of the art processors in the Texas Instruments TMS320C6400 family range up to 8000 MIPS [33]. The advantage of a DSP processor is mainly in a low power consumption, unit cost and increased memory bandwidth. Using the above estimate, we see that Turbo decoding on a processor will be limited to about 10 Mbit/s. The processor performance and decoding requirements are summed up in Table 2.1 (together with FPGA and ASIC solutions performance).

2.1.2 Performance of Hardware Solutions

The second two options mentioned above (i.e., FPGA and VLSI solutions) allow us to specify a custom structure consisting of logical elements and registers that will be implemented in hardware. Compared with *software design* it is considerably more difficult to construct a hardware structure (*hardware design*) to perform a specific task (e.g., complex decoder). Hardware description languages such as *VHDL* [2] *Verilog* [7], or *SystemC* [5] can be used. An alternative is the use of *System Generator*, what is a tool that allows us to draw up the system schematics in *Simulink/Matlab* environment.

The performance of systems designed in hardware can be substantially better that the performance of software systems even though the processing speed (i.e., the clock speed) of an FPGA or an ASIC might be substantially lower than the processing speed of an instruction based processor. Hardware design allows high *parallelization* (pipelining) which will dramatically increase the performance of the system. A decoder of a Turbo Code based on the concatenation of two convolutional codes implemented on an FPGA can process data at a rate as high as 12 Mbps [39], while utilizing only part of the logic available on the FPGA. The processing speed of a decoder of a Turbo Code based on the concatenation of block codes can perform at a rate as high as 150 Mbps.

Design intended for an FPGA can be easily ported to an ASIC, where an ASIC will generally perform better, as it does not have the disadvantage of additional delay introduced when a signal is routed through switches on the FPGA's interconnect network. Due to these advantages, an ASIC might be able to work at a higher clock frequency as well as consume approximately 20% less power.

Table 2.2: Approximate speed comparison of different FPGA speed grades

FPGA Speed Grade:	-4	-5	-6	-7
Virtex II	100%	114%	126%	
Virtex II Pro		128%	143%	161%

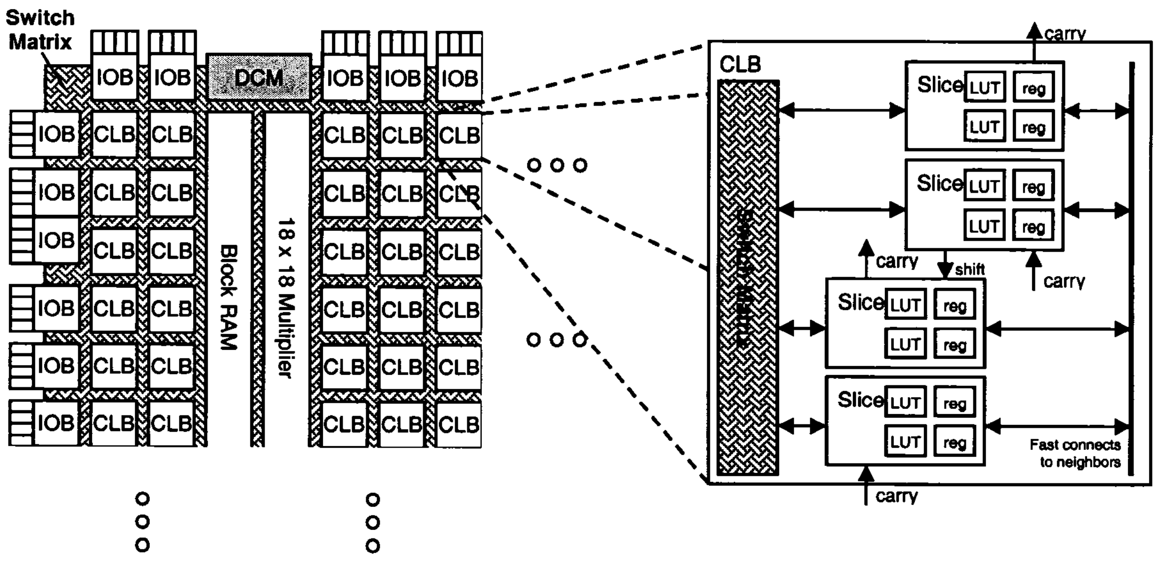
Table 2.2 lists the approximate differences in performance of the two most common FPGA families from *Xilinx*. Each of these FPGAs is manufactured in 3 different speed grades between -4 (the slowest) and -7 (the fastest). The performance is normalized to the speed of the *Virtex II* -4 speed grade chip. The table was compiled using the *Virtex II* and *Virtex II Pro* handbooks [41] [42] and comparing the claimed performance of solutions listed in [39] on different platforms. We use this table to normalize the performance of algorithms with speeds listed for one of these chips.

2.2 Hardware Design Using Programmable Logic

Due to the high throughput requirements on the algorithms in this thesis we will design these in hardware; for an implementation on an FPGA. Modern FPGAs include sufficient logic to address the complexity of these algorithms as well as built-in high speed serial interfaces that enable the FPGA to communicate with the host system at the required data rates. The following subsections describe the programmable logic technology as well as the hardware and software setup that is necessary for FPGA development.

2.2.1 Field Programmable Gate Arrays (FPGA)

An FPGA is the latest generation of programmable logic. In contrast to ASICs, the gate array structure of an FPGA is programmable after manufacture. This is achieved with an array of configurable cells that perform custom logical operation and a configurable interconnect network that is used to route signals between the cells.



(a) Matrix of elements on the chip (b) Configurable logic block (CLB) details

Figure 2.1: Virtex-II chip configuration

In this work, we will be using a system with a *Virtex-II* FPGA from *Xilinx*. Therefore, in this section, we will describe this particular chip. Nevertheless, other FPGA implementations have a similar structure. The core element, enabling configurability of the executed logic, is a 16 bit Look-Up Table (LUT). This LUT can be configured to:

- Implement a custom logic operation on 4 input bits, with the LUT storing the truth table of this operation.
- Function as a RAM block, storing a 16 bit word.
- Work as a 16 bit shift register.

In *Virtex-II*, the LUTs are grouped by two to form a Slice and four slices are grouped again to form a Configurable Logic Block (CLB). The details of a Slice are outlined in Figure 2.1(b). This grouping, with the introduction of additional signals (*carry* and *shift*), enables effective implementation of moderately complex elements (e.g., long shift registers, operations using a carry-over flag) within one CLB. The CLB is connected to a *Switch Matrix*, which enables to route signals on the FPGA chip. Additional lines connect the CLB to its neighbours in the matrix depicted in Figure 2.1(a). With elements performing elementary logic operations and a configurable network connecting these elements, we are able to emulate the behaviour of any logic circuitry.

Input/Output Blocks (IOBs) placed on perimeter of the chip connect the FPGA to external signals via an actual physical pin of the chip. Using the *Switch Matrix*, these external signals can be routed from the IOB to any other location on the FPGA. Other resources, such as 18 by 18 bit multipliers and RAM blocks, are available to the designer and accessible using the *Switch Matrix*. The programming of an FPGA consists of:

- Configuring the CLBs to perform synchronous (employing the register *reg* in Figure 2.1(b)) or combinatorial operations.
- Setting-up the *Switch Matrix* to properly route signals between the CLBs and other resources on the FPGA.

The additional logic in these elements that is necessary to allow the programming of the chip causes an FPGA to be slower and less power efficient than an equivalent ASIC. The advantage of an FPGA is the shorter time-to-marked and the lower development cost which makes ASIC design feasible only for large quantities (above ten thousand units). Nevertheless, if desired an ASIC can be made that is a hard-copy of an FPGA.

2.2.2 Hardware Design for an FPGA

To design an algorithm on an FPGA, we have to define the behaviour of the chip. We have three possible entry points for hardware design:

- Using a *Hardware Description Language* (HDL), e.g., *VHDL*, *Verilog*, *SystemC*,
- Describing the design using schematics,
- Implementing the behaviour in a high-level sequential language, such as *Java*.

Specifications, in any of these formats, must be compiled into a *net list*. A *net list* specifies how to implement the design using elementary logic by enumerating the logic elements and listing the connections between these elements. This specification is then mapped to the resources of an actual FPGA. A binary file, called *bitstream*, is created and this file can be used to program the FPGA.

Creating a *net list* from a schematic specification or a specification in *VHDL* or *Verilog* is a straightforward, even though not simple, process. On the other hand, creating a description suitable for hardware from a high-level sequential language, which was not designed for hardware development, is tricky. Several utilities, which enable a translation from a sequential language to *VHDL* or *Verilog,* are available on the market. An example is *Forge* [38] developed by *Xilinx*, which facilitates the translation from *Java* to *VHDL.* Nonetheless, these utilities do not produce optimal results and are scarcely used by hardware designers.

The way programming is uploaded onto a chip and the way an FPGA chip can communicate with the surrounding environment are dependent on the development board that contains the chip. Generally constraints specifications have to be created to define which pin of a FPGA chip will be driven by which internal signal in the design. These signals can than drive other logic present on the board alongside the FPGA (e.g., RAM, digital to analog converters, PCI/USB controllers). Often there is a communication controller present on the board such as the previously mentioned PCI or USB controller that allows us to transfer data between the FPGA board and a host PC. After transferring the design to the FPGA board, we can verify its functionality by stimulating it appropriately and monitoring the response.

2.2.3 Hardware Design Using a Hardware Description Language (HDL)

A *Hardware Description Language* such as *VHDL*, *Verilog* or *SystemC* describes the behaviour of a hardware circuit. This description can be used for:

- Simulation, where we try to verify the functionality of the circuit and
- Synthesis, the process of mapping the description onto the hardware.

In this work, *VHDL* is used internally by the tools involved in our design process. This language is also used to integrate third party *intellectual property* (*IP cores*), such as a Reed-Solomon encoder/decoder, into our designs. Thus, let us introduce the language in brief. In the following source code we present the *VHDL* implementation of these three various elements: an *and gate*, a *d-latch* and a *4-bit register.*

A *d-latch* is a simple circuit that samples the value of its *q* input when the *clk* input is high and otherwise holds the value of its *q* output. Each of these implementations demonstrated a different construct in *VHDL*. The *d-latch* is implemented using a behavioural model that describes the behaviour of the element in sequential statements similar to a high-level language. The *and gate* is described using a functional model that consists of concurrent assignment operations. The *4-bit register* is then constructed using four *d-latches* and a *and gate* and connecting the inputs and outputs of these elements. This description is called a structural model. Please refer to referenced literature for further information on *VHDL*.

VHDL source code: Example of using behavioural and structural models of entities

```
--------------------------------------
-------------- D LATCH -------------

entity d_latch is
    port(d,clk : in bit; q : out bit);
end d_latch;

--------- Behavioural model ---------

architecture behav of d_latch is
begin

    p0 : process is
    begin
        if clk = '1' then
            q <= d after 2 ns;
        end if;
        wait on clk, d;
    end process p0;

end architecture behav;

--------------------------------------
------------- AND GATE -------------

entity and_gate is
    port(a,b : in bit; y : out bit);
end and_gate;

--------- Functional model ---------

architecture funct of and_gate is
begin

    y <= a and b after 2 ns;

end architecture funct;
```

```
--------------------------------------
---------- 4 BIT REGISTER ----------

entity reg4 is
    port(d0,d1,d2,d3,en,clk : in bit;
         q0,q1,q2,q3 : out bit);
end entity reg4;

--------- Structural model ---------

architecture structural of reg4 is
    signal int_clk : bit;
begin

    bit0 : entity work.d_latch(behav)
        port map (d0, int_clk, q0);
    bit1 : entity work.d_latch(behav)
        port map (d1, int_clk, q1);
    bit2 : entity work.d_latch(behav)
        port map (d2, int_clk, q2);
    bit3 : entity work.d_latch(behav)
        port map (d3, int_clk, q3);
    gate : entity work.and_gate(funct)
        port map (en, clk, int_clk);

end architecture structural;
```

Figure 2.2: VHDL source code example

The *VHDL* description can be simulated using a *HDL* simulator such as *ModelSim* [21], which can be supplied with input test vectors and compare the outputs against expected output sequences.

2.2.4 From Hardware Description to Hardware Programming

To translate the design onto an FPGA, another tool such as *XST* (which is part of the *Xilinx* integrated development environment *ISE* [40]) has to be employed. This elaborates the description and creates a *net* consisting of the interconnection of elementary logical elements. This *net* fully specifies the logic that is necessary to implement the functionality, as described by the *VHDL* source code.

With the *net* already elaborated, the logic elements have to be mapped onto the resources available on the FPGA. This process is called *place and route.* Each logic elements in the *net* is assigned to a specific resource on the *FPGA* and the configuration of the *Switch Matrix* (internal signal routing of the FPGA) is determined to properly route all the signals.

The *place and route* process might fail if there are not enough resources on the target FPGA to either implement the specified logic or to route the signals. After the design has been placed onto the *FPGA*, the maximum clock speed can be determined by computing the maximum register-to-register delay. Typically only about half of this delay would be due to the design logic. The other half would be due to the delay introduced by the switches on the *FPGA*'s *Switch Matrix.* If we are not satisfied with the achieved clock speed, the tools can try to alter the placement and routing to cut down the biggest delay.

The *synthesis* as well as *place and route* processes are executed by different tools that are all integrated into the development environment from Xilinx.

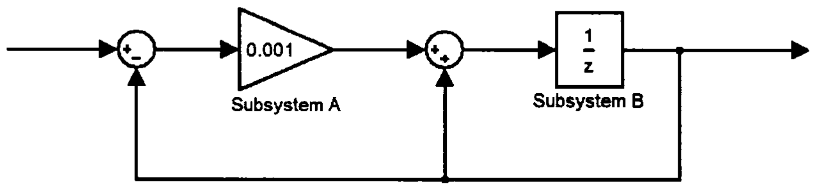
2.2.5 Using System Generator in Hardware Design

System Generator from *Xilinx* [43] is a graphic tool that allows us to draw up the schematics of the desired system and is based on the *Matlab/Simulink* [20] environment, a model-based environment for simulation of dynamic systems. The model consisting of interconnected blocks can be translated directly onto an

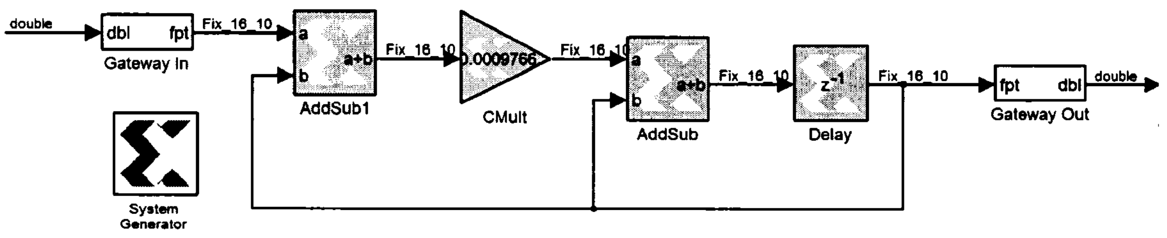
FPGA. *System Generator* uses a graphic interface where blocks (subsystems) are represented by icons and signals by lines connecting these icons. The diagram in Figure 2.3(a) is an example of a simple *Simulink* system.

System Generator provides us with blocks representing basic hardware elements (e.g., registers, logic gates, math operations, RAM) in *Simulink.* If we build a circuit from these blocks, we can simulate it inside *Simulink* to determine its behaviour. We can use this simulation to verify the functionality of the design without having to compile it into a hardware programming file. *Simulink* provides us with multitude of tools to analyze data generated by this simulation. If the behaviour conforms to what we request of the system then we can use *System Generator* to translate the schematics into hardware. The diagram in Figure 2.3(b) shows the implementation of the system from part (a) using *System Generator* blocks.

Moreover, all its blocks have an equivalent FPGA implementation. A circuit designed using *System Generator* can be easily translated into FPGA programming and the translation is done automatically in three steps:



(a) *Simulink* model of a system with feedback



(b) The system in (a) implemented using built-in *System Generator* blocks

Figure 2.3: Transforming a *Simulink* model into *System Generator* design

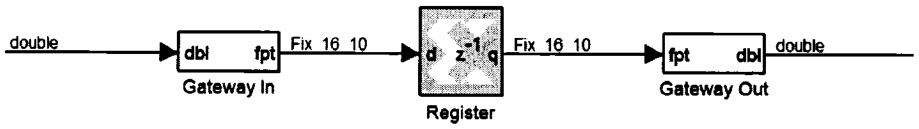
1. *VHDL* source code and *Xilinx* development environment configuration files are created to instantiate all the logic elements in the schematics and to connect the elements properly.
2. The generated source and project files are compiled into an FPGA programming file (i.e., a *"bitstream"*).
3. A hardware co-simulation block is created in *Simulink*. This block represents the compiled design. If the block is included in schematics, *System Generator* will automatically upload the design to an FPGA, where it will be executed in parallel with the *Simulink* simulation.

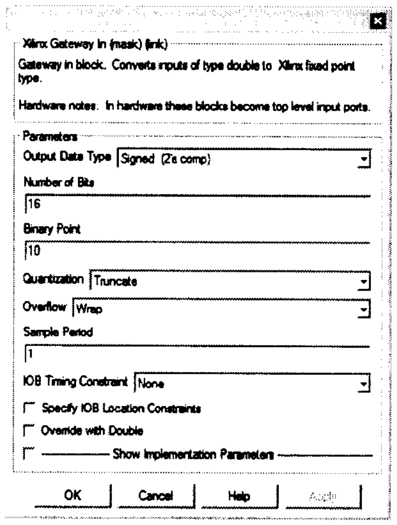
In the last step, the design running on hardware was integrated back into *Simulink* environment. This is useful for hardware co-simulation, but the design can also be uploaded into the FPGA independently for standalone solutions.

Integration of *System Generator* and *Simulink* design

The signal interconnects in Figure 2.3(b) represent hardware busses which have to be fixed point binary numbers in simulation. General *Simulink* blocks cannot be translated into hardware (there is not an equivalent hardware implementation for every possible *Simulink* block) and, furthermore, the *System Generator* blocks only accept fixed point numbers as inputs, whereas a general *Simulink* signal is a floating point double precision number (*double*). Therefore, any *Simulink* design using other block libraries than the ones provided by *System Generator* has to be isolated using translation blocs. This design will not be translated into hardware.

The translation is done using the *Gateway In* and *Gateway Out* blocks shown in Figure 2.3(b). The *Gateway In* block must specify the format of the input. In Figure 2.4, the output of a *Gateway In* is specified (in the parameters dialog box) to be a signed 16 bit number with 10 decimal bits. Usually the format of the output signal of a block can be inferred from the format of the input. For example, in the same figure the output of the "Register" block will be of the same format as its input signal. The output of a *Gateway Out* will be *double*, independent on the format of the input, as the *Gateway Out* marks the transition back to the *Simulink* environment. The gateway blocks also allow specifying to which actual pins of the *FPGA* chip these signals will be routed (IOB Location Constraints), when translated into hardware programming.





Xilinx Gateway In (mask) (link)

Gateway in block. Converts inputs of type double to Xilinx fixed point type.

Hardware notes: In hardware these blocks become top level input ports.

Parameters

Output Data Type: Signed (2's comp)

Number of Bits: 16

Binary Point: 10

Quantization: Truncate

Overflow: Wrap

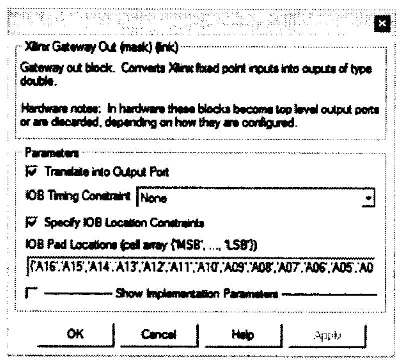
Sample Period: 1

IOB Timing Constraint: None

☐ Specify IOB Location Constraints

☐ Override with Double

☐ Show Implementation Parameters



Xilinx Gateway Out (mask) (link)

Gateway out block. Converts Xilinx fixed point inputs into ouputs of type double.

Hardware notes: In hardware these blocks become top level output ports or are discarded, depending on how they are configured.

Parameters

☑ Translate into Output Port

IOB Timing Constraint: None

☑ Specify IOB Location Constraints

IOB Pad Locations (cell array {'MSB', ..., 'LSB'})

{'A16','A15','A14','A13','A12','A11','A10','A09','A08','A07','A06','A05','A0

☐ Show Implementation Parameters

Figure 2.4: Illustration of the concept of signal translation from the floating point domain to fixed point domain using *Gateway In* and *Gateway Out*

Hardware co-simulation

When finishing the compilation process, *System Generator* creates a block that represents the programming of the *FPGA* and is associated with the new *bitstream*. If we include this block in a *Simulink* design, the *bitstream* will be uploaded onto the *FPGA* and it will run in hardware. The inputs and outputs of the design will be connected to *Simulink* signals as specified. Data can be transferred between the *Simulink* simulation and the design running on the *FPGA* via an appropriate interface (e.g., JTAG, USB, PCI).

Nevertheless, we have to be aware that *Simulink* runs at a much slower "clock" speed than the *FPGA*. Thus these two designs (one being simulated in *Simulink* and the other running on the *FPGA*) run on independent clocks and thus the communication between them has to be handled as communication between two asynchronous circuits. A setup like this is very useful if we use the *FPGA* to accelerate simulations under different conditions. We call this concept *hardware co-simulation.*

While we want the simulation to run at full hardware speed of the *FPGA*, we want to be able to easily influence the configuration of the simulation parameters. Therefore, it is better to generate the configuration parameters inside *Simulink*, where we have more flexibility and it is not necessary to recompile the whole design to introduce changes.

Internally, *System Generator* creates a register for each *Gateway In* and *Gateway Out* block. This register is used during hardware co-simulation to realize the communication between the design running in the FPGA and the *Simulink* environment. *System Generator* integrates custom logic into the FPGA that enables it reading and writing such registers using a register transfer protocol. This protocol enables the host PC connected to the FPGA board to retrieve and set the values of the inputs and outputs of the designed logic. In the host PC, the *Gateway In* and *Gateway Out* blocks use customized *System Generator* code to access the register transfer protocol and integrate the hardware registers into the software simulation in *Simulink.*

The hardware link used for the register transfer protocol is dependent on the development board configuration, where theoretically any data interface connecting the board to the host PC can be used. *System Generator* generally supports only the JTAG interface while other interfaces would require custom logic in the FPGA and customization of the *System Generator* gateway blocks. Nevertheless, virtually all development boards have the JTAG interface and this makes *System Generator* compatible with virtually any *Xilinx* based FPGA development board. Only a small configuration file is required by *System Generator* to set-up the communication via the JTAG interface for hardware co-simulation. The disadvantage of using the JTAG is that it is fairly slow and therefore the data transferred into *Simulink* from the FPGA board will be updated at a fairly slow rate.

System Generator supports other interfaces (USB or PCI) on a few specific boards. One of the boards supported by *System Generator* is the *Xtreme DSP Kit* development board from *Nallatech.* This development board has both: a USB and a PCI interface. The vendor provides hardware design and software API that implements a register transfer protocol between the board and the host PC via one of these interfaces. *System Generator* uses this design and API to enable hardware co-simulation on this specific board using the USB or PCI interface. Thus enabling faster communication with the board and improving the performance of the hardware co-simulation scheme.

Customizable design using Simulink

In our design paradigm, we want to develop a front-end customizable communication library, where all parameters must be configurable in the front-end interface. In a hierarchical design, this requires that the parameters are propagated downwards through the design hierarchy up to the point where the parameter invokes the necessary customization. In some cases, this customization might involve only setting a specific parameter of a *System Generator* block. In other cases, this customization might require custom processing, where different design has to be instantiated depending on the value of the parameter. This custom processing has to be defined within a module and encapsulated into the module's front-end interface. The complexity of the custom processing might require the employment of a programming language. The *masking* methodology in *Simulink* enables both: to propagate parameters down a design hierarchy and to apply customization using a powerful programming language – Matlab.

The masking concept assigns a parameter dialog box to a *Simulink* block and hides the internal design of the block from user view. The masks of the *Gateway In* and *Gateway Out* blocks in Figure 2.4 are good examples of the mask dialogs. Parameters set within the mask can be used in the block's design to either initialize the block using *Matlab* scripts or to set values within the masks of sub-blocks contained within the block. This way, parameters can propagate through a hierarchy of masked blocks. An initialization script that is executed whenever the design is updated can be defined for each masked block. The design is updated before a simulation is run and before it is translated into hardware.

2.3 Commercially Available Solutions

Numerous commercially available hardware designs can be included in your system on an FPGA or an ASIC. These are usually precompiled designs with a documented outward interface and are called IP-cores (Intellectual Property

Cores). Implementations of FEC are also available. Whereas these cores are usually optimized and it would be hard to compete with their performance, the cores are implemented according to an industrial standard and implement a very specific code with no or little possibility of customization (e.g., in terms of rate, code parameters, soft-decision decoding abilities). Table 2.3 lists some FEC solutions available from Xilinx. The speeds are derived from the datasheets of these cores and normalized to the performance of a -4 speed grade *Virtex II* FPGA chip according to Table 2.2.

2.4 Initial Platform Setup

Our initial platform setup consists of a design environment as well as the FPGA hardware and is the starting point for a prototyping platform described in the Chapters 3, 4 and 5. The prototyping platform augments this initial setup to enable rapid prototyping of error control codes and to allow emulation of a coded optical CDMA network.

Table 2.3: Parameters of error-control solutions available commercially from Xilinx

Algorithm	Speed	SLICES	Block RAM
Reed Solomon			
(207,187) 8-bit	722 Mbps	764	2
(255,239) 8-bit	727 Mbps	877	4
Viterbi Decoder[1]			
Parallel, cl=5, tb=30, sb=3	150 Mbps	805	2
Parallel, cl=7, tb=42, sb=4	132 Mbps	3299	4
Parallel, cl=9, tb=54, sb=4	91 Mbps	12821	16
Serial, cl=5, tb=30, sb=3	18 Mbps	261	2
Serial, cl=7, tb=42, sb=3	15 Mbps	638	2
Serial, cl=9, tb=54, sb=3	10 Mbps	2107	4
Turbo Product Code (64,57)[2], 5 iterations	123 Mbps	3313	17
Turbo Convolutional Code[2], 5 iterations	6.2 Mbps	1618	47

[1] cl: constraint length, tb: trace-back length, sb: soft value bit width
[2] 3G compliant, sliding window size: 32, block size: 378 – 20730
Reference: Xilinx, http://www.xilinx.com/ipcenter/fec_index.htm [39]

Figure 2.5: The Xtreme DSP Kit development board

We have chosen *System Generator* from *Xilinx* as the front end design interface of our platform because of the following reasons:

- It is visually attractive, easy to learn and allows us to build modules with clear interface.
- It enables us to use Matlab as a powerful scripting and specification language in applying customization into the hardware design.
- It facilitates a "push of a button" transition to a hardware programming file.
- It enables easy and intuitive communication with the running design seamlessly integrated into *Simulink*.
- It supports most *Xilinx* based *FPGA* platforms.

System Generator offers a hardware design environment that is compatible with almost all *Xilinx* based development boards with data exchange between *System Generator* and the board (necessary to conduct hardware co-simulation) via the JTAG interface. Nevertheless when using JTAG, data can only be transferred at a fairly low rate and this configuration could influence the performance of our simulation testbed.

For use in our platform, we have chosen the *Xtreme DSP Kit* development board from *Nallatech* in Figure 2.5. The board hosts either an *XC2V2000* or an *XC2V3000 Virtex II* FPGA chip. The resources available on these devices are outlined in Table 2.4. Moreover, this board allows us to use the communication

Table 2.4: Various FPGA devices resource comparison

FPGA Device	SLICES[1]	Block RAM[2]	Multipliers
XC2V2000	10,752	56	14
XC2V3000	14,336	96	16
XC2V8000	46,592	168	28

[1] 1 SLICE = 2 LUT, 4 SLICES = 1 CLB

[2] Each Block RAM has 18kbits

via the USB supported in *System Generator.* This setup is to be used for hardware co-simulation while testing the performance of the designed error control algorithms or when emulating the optical network within the FPGA and it does not provide sufficient host-to-FPGA bandwidth for the required speeds. Other hardware options (e.g., Avnet's Virtex-II Development Kit, Xilinx ML310 Embedded system development board) are considered for our prototype deployment, where constraints on connectivity with an optical link require a board with a high-speed digital interface. Nevertheless, these development boards do not have a high speed interface that could be utilized by *System Generator* to perform hardware co-simulation and therefore are not suitable in the initial phase of our project.

Several other software packages have to be installed on a workstation to facilitate design in *System Generator* and communication with the *Nallatech Xtreme DSP Kit.* These packages include *Matlab*, *Xilinx ISE* and *Nallatech* software suite. The detailed instructions, on how to set-up a workstation, are included in Appendix A.

2.5 Chapter Summary

In this chapter, we have introduced the software and hardware setups of our development platform. We have also outlined other design options to give support to our decisions and for reference purposes.

Chapter 3

Proposed Modular Design Architecture

This chapter and the following Chapter 4 introduce design methodologies and concepts that are employed in the development of our rapid prototyping platform described later in Chapter 5. These design methodologies were drawn up to:

- Allow the construction of front-end customizable communication modules,
- Enable easy data routing and scheduling for data processing,
- Facilitate clear, fast and error-free design.

The prototyping platform incorporates several concepts that had to be designed to allow for easy usage, high customizability, convenient design verification and runtime control. These concepts are:

- Self-descriptive packets to enable easy routing and scheduling of data processing,
- A unified module interface and internal naming conventions,
- A Generate block that allows to generate arrays of regular hardware structure enabling high level customization,
- A system of "debugging" modules to verify the design operation,
- A *Hardware-Simulink-Matlab* interface to retrieve data from a system running on the hardware, control it and run batch-tests.

The following sections in this chapter describe the first two bullets of the above concepts while the remaining concepts are explained in detail in Chapter 5. First, we will discuss the concepts and advantages of our modular design. Next, we will describe how to implement a specific module based on these concepts. Finally, we present a simple example of an iterative loop employing modules based on our concepts that demonstrates its advantages in data routing and parallelization.

The implementation of our platform is done in *System Generator* from *Xilinx* due to its advantages outlined in Section 2.4.

3.1 Module Data Exchange: Packets

Communication is a problem of reliably delivering information from the *source* to the *sink* through a *channel*. In digital communication, an *encoder* and a *decoder* allow increased reliability of information transmission by introducing redundancy into the transmission and formatting the message for channel impairments. We commonly draw a diagram of such a communication system that looks like the diagram in Figure 3.1.

Such a diagram clearly states the components involved in a communication systems or the flow of information and dependencies between these components. In our hardware design paradigm, we would like to work with the same level of abstraction and simple interface as used in the above diagram. Commonly, more complicated interfaces are used between modules that include several control signals going in both directions.

This is not only aesthetically displeasing, but also creates opportunities to introduce errors into the design. With a feedback loop via the signals in both directions neither of the modules A and B can be tested properly without the other one and more care has to be taken to properly anticipate all situations that can arise on the interface of these two modules.

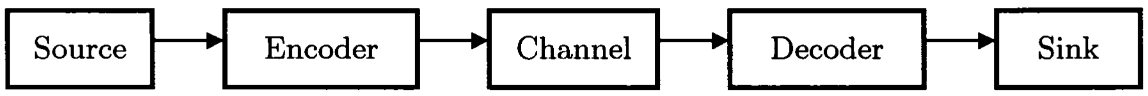


Figure 3.1: Schematic block diagram of a digital communication system

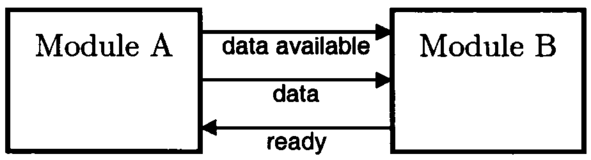


Figure 3.2: Data interface with handshaking

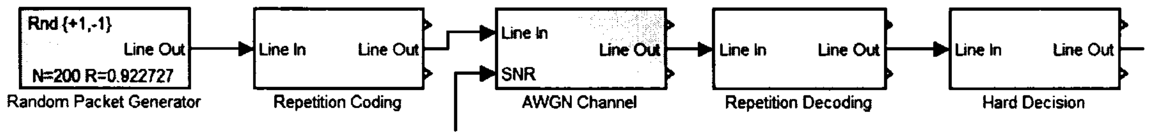


Figure 3.3: Digital communication system as specified using our library modules

In the library we implement a scheme where data is transmitted on a single line (even though this bus is a multi-bit bus it is not being considered to consist of multiple control signals) in packets that are self-descriptive. That is:

- The meaning of the data (packet type) can be automatically identified.
- Data boundaries and null transmission times can be identified.

Furthermore, all modules in the library are required to be always ready to accept a packet on the input but they may route this packet to a special "Fall Through" output if the module is:

- Not ready to process the packet,
- Not able to process the specified *packet data type.*

These two concepts eliminate the necessity for additional signalling (and notably they eliminate the need for feedback) between two modules, thus keeping the interface as simple as in the previous abstract block diagram of a communication system. Figure 3.3 shows a setup of a generic communication system that can be used for performance testing using modules from our library. Note that the "Fall Through" outputs of the modules are left unconnected.

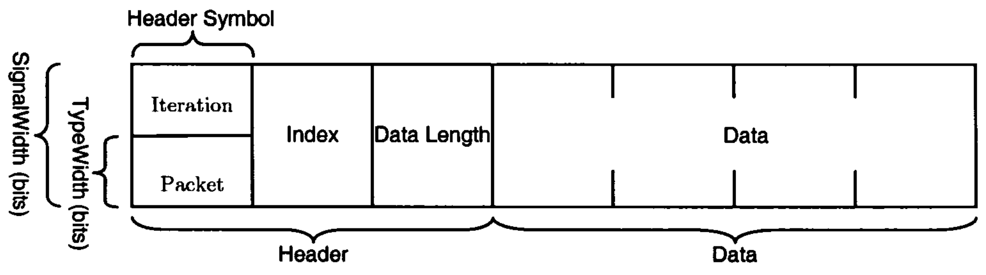


Figure 3.4: Data-packet structure

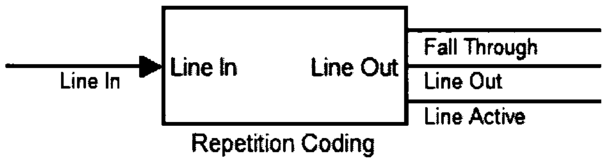


Figure 3.5: An example of a module

3.1.1 Packet Structure

A *data line* is a k-bit bus that connects two modules. *Packets* can be transmitted on this data line and if no packet is being transmitted, all the bits of the *data line* have to be set to 0. Thus, the beginning of a packet is recognized as the first nonzero symbol on the line. The packet starts with a header that is 3 symbols long and specifies the packet type, some user information and the packet length **L**. The next **L** symbols are the payload data of the packet (thus also zero symbols will be identified as packet data). After the payload data has been transmitted, the line returns to all-zero state or a nonzero symbol that would be identified as the beginning of a new packet. Figure 3.4 details the structure of a typical packet.

The packet header consists of 3 symbols: *header symbol*, *index* and *data length.* The *header symbol* consists of two parts: the *packet type* and the *iteration.* The *packet type* is a number (user defined code) that should identify the meaning of the data. This information can be used by modules to selectively process packets. *Packet type* has to be nonzero, as this guarantees that the first header symbol of a packet is nonzero and that the start of a packet will be properly recognized. The *iteration* is a number that can be increased whenever a packet is processed by a module. Thus this number can be used if a packet should cycle through a module several times.

The *index* symbol should be used for packet indexing (identification). Thus if packets are out of order after some processing, this number can be used to reorder them. Nonetheless, this symbol can be also used otherwise if necessary. The *data length* defines the length of the data portion of the packet. As this symbol has the same width as the line, the maximum data length is 2^k-1. For additional information on packet composition refer to Table B.2 in Appendix B.

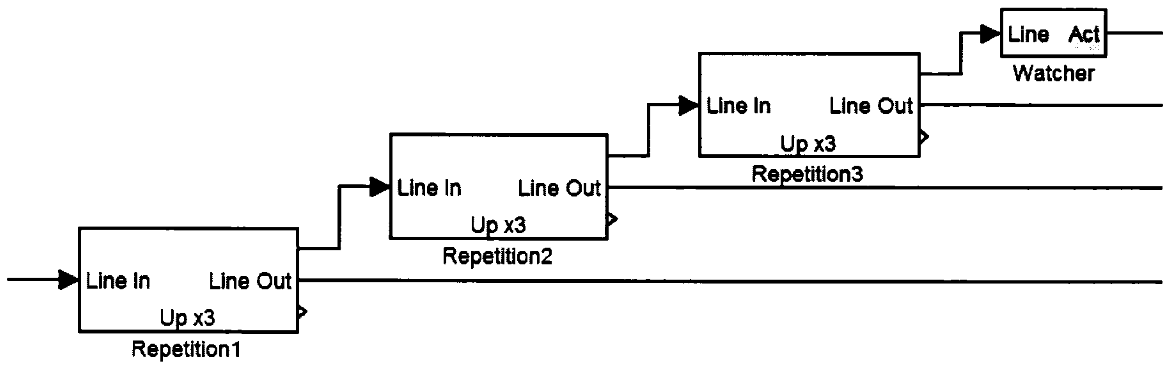
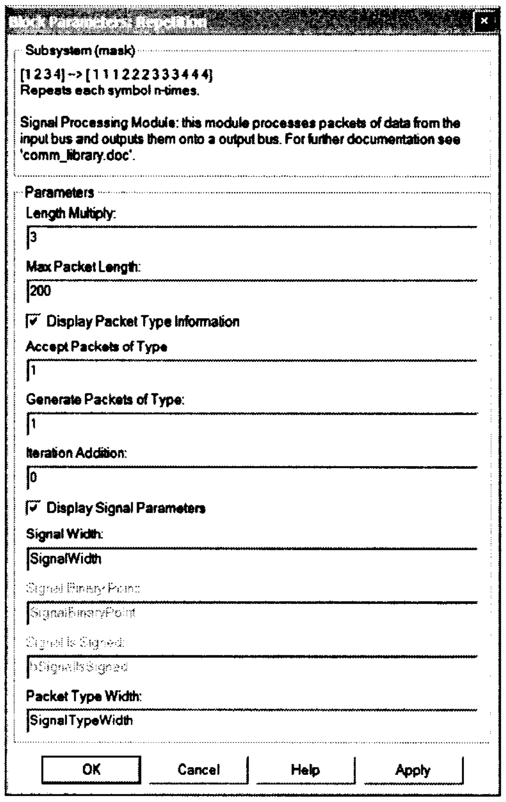


Figure 3.6: Linking modules

3.1.2 Module Structure

A module processes packets from the "Line In" input and outputs processed packets to the "Line Out" output. If a module cannot process a packet then it outputs the packet on the "Fall Through" line. The third output, "Line Active", is a signal that is high when there is packet being transmitted on "Line Out". This output is provided mainly for aesthetical reasons – to keep the module symmetric. A module does not process a packet either if it has a wrong *packet type* or if the module is busy. For example, a rate $1/3$ repetition coding module codes a packet with 200 data symbols to a packet with 600 data symbols by repeating each symbol 3 times. Thus after receiving the 200 data symbols on the input line it will stay busy for another approximately 400 clock cycles. Therefore, if the packets come one after another on the input line, this module will not be able to process them. It is possible to link multiple modules by connecting another repetition encoder to the "Fall Through" line of the previous module, as depicted in Figure 3.6, and thus achieve (if one links three repetition encoders) that all packets will be processed. A *Watcher* module can be used to check for any activity on the *Fall Through* output of the last repetition encoder. If there is activity, congestion occurred and the loss of a packet can be reported. Nonetheless, in our example, the three repetition modules will be able to process all incoming packets and congestion will never occur.



Block Parameters: Repetition

Subsystem (mask)

[1 2 3 4] --> [1 1 1 2 2 2 3 3 3 4 4 4]
Repeats each symbol n-times.

Signal Processing Module: this module processes packets of data from the input bus and outputs them onto a output bus. For further documentation see 'comm_library.doc'.

Parameters

Length Multiply: 3

Max Packet Length: 200

[✓] Display Packet Type Information

Accept Packets of Type: 1

Generate Packets of Type: 1

Iteration Addition: 0

[✓] Display Signal Parameters

Signal Width: SignalWidth

Signal Binary Point: SignalBinaryPoint

Signal Is Signed: bSignalIsSigned

Packet Type Width: SignalTypeWidth

Figure 3.7: Mask of a Repetition Module

3.1.3 Module Parameters

Each module has a set of parameters that are unique to that modules and a set of common parameters. All these parameters are brought to the front-end interface of the module. In *Simulink*, this front-end interface is called a *mask* (from the fact that it is "masking" the internal structure of the module/block) and it is represented by a dialog. In Figure 3.7, we can see an example of a mask of a *Repetition Coding* module.

When we use a module in a design, we no longer have access to its internal structure. All customizations to the module have to be done through setting the *mask* parameters. This strictly modular design paradigm allows for easy reusability of implemented designs and decreases the probability of errors. Nevertheless, depending on the parameters, the module can modify its own structure by means of built-in scripts which are hidden from the user.

The common module parameters include parameters describing:

- Packet processing,
- "Line In" specification and interpretation,
- "Line Out" specification and interpretation.

Packet processing parameters describe:

- Processed packet types (*Accept Packets of Type*); packets that do not have the specified *Packet Type* will be routed to the "Fall Through" output,
- The *Packet Type* of the output packet (*Generate Packets of Type*),
- The increment to the *Iteration* number in the packet header (*Iteration Addition*).

"Line In" / "Line Out" parameters specify the width of the bus (*Signal Width*), the width of the *Packet Type* component inside the first packet header symbol (*Packet Type Width*) and the numeric interpretation of data symbols (*Signal Binary Point* and *Signal Is Signed*). Note that some parameters might be left unused in certain modules. In the example in Figure 3.7, the numeric interpretation of the data symbols is irrelevant, because a symbol can be repetition-encoded without the knowledge of its interpretation. Consequently, these parameters are disabled. Similarly, the "Line Out" parameters of most modules are usually identical to the "Line In" parameters. Thus these parameters are included only if the module changes the structure of the packets. For additional information on module *mask* parameters, please refer to Appendix B.

3.2 Module Implementation Methodology

This section is dedicated to the implementation of a module that conforms to the specifications of the "outward" interface from the previous section. Internally all the *mask* parameters are variables that are available and can be used in the module initialization procedures, passed to other masked subsystem or used directly to customize Xilinx blocks inside the design. We use a naming convention for these *mask* variables to distinguish them from the other variables in the various *Matlab* contexts. All *mask* parameters are prefixed with '*p_*', where *p* stands for *parameter*. Corresponding to parameters in the *mask* the following variables are defined in all modules (for details, please refer to Appendix B):

- p_AcceptPacketType, p_ProcessedPacketType, p_IterationAddition,
- p_SignalWidth, p_SignalBinaryPoint, p_bSignalIsSigned, p_SignalTypeWidth.

These variables are defined in modules that alter the signal structure:

- p_OutputSignalWidth, p_OutputSignalBinaryPoint, p_bOutputSignalIsSigned, p_OutputSignalTypeWidth.

Further parameters are defined in individual modules and the following text we will address these implementation issues:

- The structure of the module (implementation of the module's logic),
- Parameter passing down through the module hierarchy,
- Module's *Simulink* mask (implementing the *Simulink/Matlab* module behaviour).

3.2.1 Module Internal Structure

Figure 3.9 details the internal schematics of a *Symbol Conversion* module. This module is used to convert symbols between two packet lines of different type. The module does not perform the conversion of the packet data symbols, but instead it provides an output "s out" and input "s in" where the user of the block connects the logic that does the necessary conversion. Figure 3.8 shows a *Symbol Conversion* module implementing a simple *bit slice.*

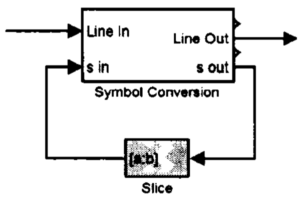


Figure 3.8: *Symbol Conversion* module

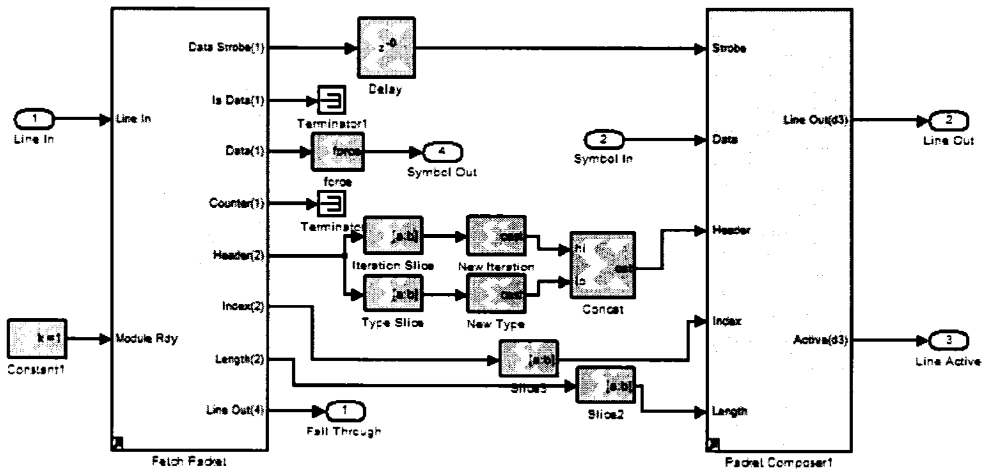
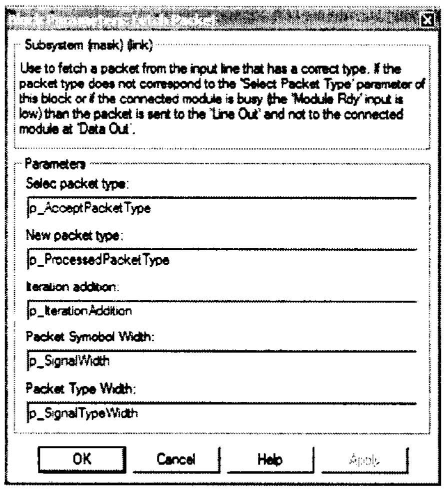


Figure 3.9: Schematics of the ***Symbol Conversion*** module

Consequently, the *Symbol Conversion* module contains a minimum of custom logic and its internal components illustrate the structure of a general module. A module generally consists of three parts:

- A block responsible for parsing the packet from the input line,
- The module's custom logic,
- A block responsible for composing a packet on the output line.

In Figure 3.9, the *Fetch Packet* and *Packet Composer* blocks are responsible for parsing the packet from the line and composing it onto the output respectively. In this simple


Subsystem (mask) (link)

Use to fetch a packet from the input line that has a correct type. If the packet type does not correspond to the 'Select Packet Type' parameter of this block or if the connected module is busy (the 'Module Rdy' input is low) than the packet is sent to the 'Line Out' and not to the connected module at 'Data Out'.

Parameters

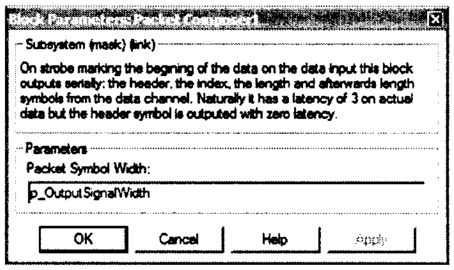
Selec packet type:
p_AcceptPacketType

New packet type:
p_ProcessedPacketType

Iteration addition:
p_IterationAddition

Packet Symobol Width:
p_SignalWidth

Packet Type Width:
p_SignalTypeWidth


Subsystem (mask) (link)

On strobe marking the begining of the data on the data input this block outputs serially: the header, the index, the length and afterwards length symbols from the data channel. Naturally it has a latency of 3 on actual data but the header symbol is outputed with zero latency.

Parameters

Packet Symbol Width:
p_OutputSignalWidth

Figure 3.10: Setting the ***mask*** of ***Fetch Packet*** and ***Packet Composer*** modules

illustration, the custom logic in between these two blocks connects their corresponding inputs and outputs while only adjusting the bit-widths of the signals using a *Slice* block. The processing of the data symbols, which must be implemented "in-between" the *Symbol Out* and *Symbol In* port blocks in Figure 3.9, is left to the user of the module.

The *Fetch Packet* block recovers the packet from the line if the packet has the *packet type* specified by *p_AcceptPacketType.* To do this, we have to pass this variable to the *Fetch Packet* block by entering the variable into its *mask*, as seen in Figure 3.10. Similarly other parameters are passed to the *Fetch Packet* module and other blocks inside any module. In fact, *Fetch Packet* again consists of multiple masked blocks, and thus, this block similarly gives a variable name to the value that was passed to it and passes it to the next level. This is the methodology that brings all module parameters to the outermost interface. Once a value is assigned to a parameter in the module's *mask* dialog box, this value is propagated via a tree of variables to all sub-blocks that need the specific parameter.

Once *Fetch Packet* module identifies a packet that should be processed, it outputs the data content of the packet and produces a set of control signals. These outputs are:

- 'Data', the data content of the packet,
- 'Data Strobe' signals the start of the data on the 'Data' output,
- 'Is Data' is high while there is valid data on the 'Data' output,
- 'Counter' is a down-counter that gives the index within the packet of the data currently on the 'Data' output,
- 'Header', 'Index', 'Length' keep a constant output that is equal to the value of the three respective header symbols of the whose data is outputted on the 'Data' output,
- 'Line Out', the *Fall Through* line where packets that cannot be processed are routed.

The *Packet Composer* block is the reverse of the *Fetch Packet* block. It accepts multiple control signals and a data bus as inputs and has a single output that is the packet line. When the 'Strobe' input goes high the block expects the first data symbol of the packet to be at the 'Data' input. Then it constructs the packet header (according to the three packet header inputs) and appends the data coming in at the 'Data' input to it and outputs the constructed packet on the 'Line Out' output.

In Figure 3.9 of the schematics of the *Symbol Conversion* module, we see that the control signals produced by *Fetch Packet* directly match with the control inputs of the *Packet Composer*. Yet, depending on the latency of the user logic, the 'Data Strobe' signal is delayed so that the 'Strobe' input and the data on the 'Data' input of the *Packet Composer* are correctly aligned.

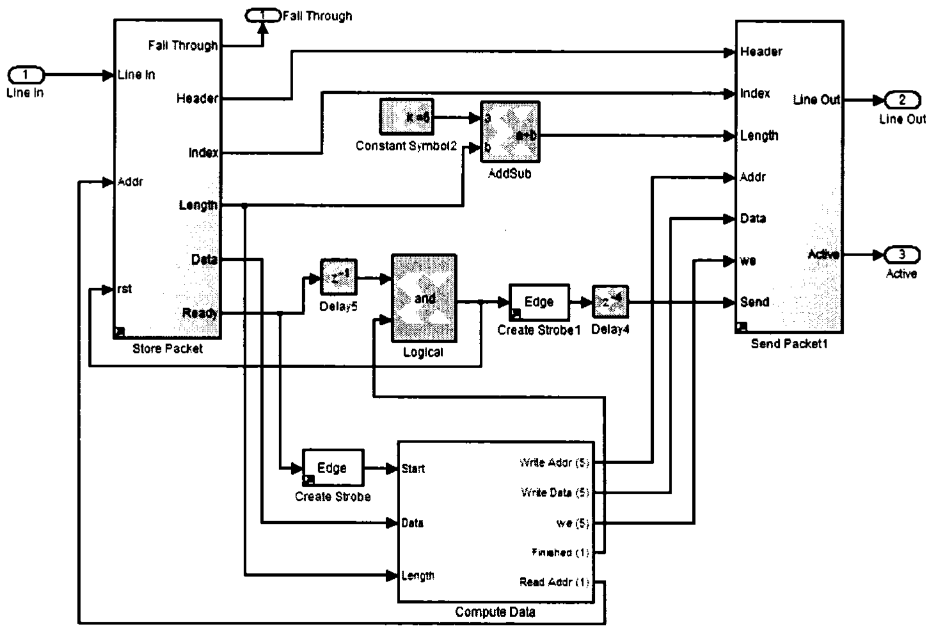


Figure 3.11: Schematics of the *FIR Filter* module

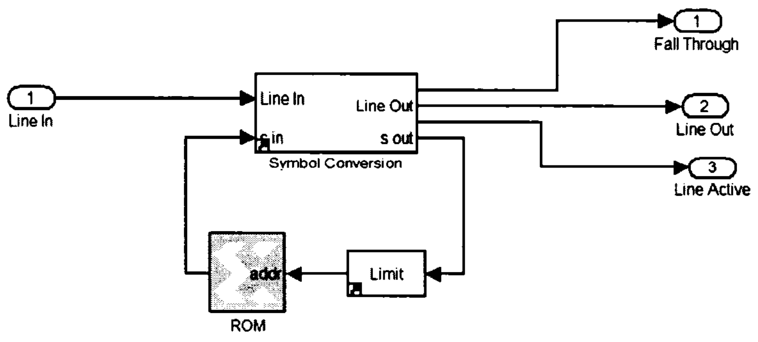
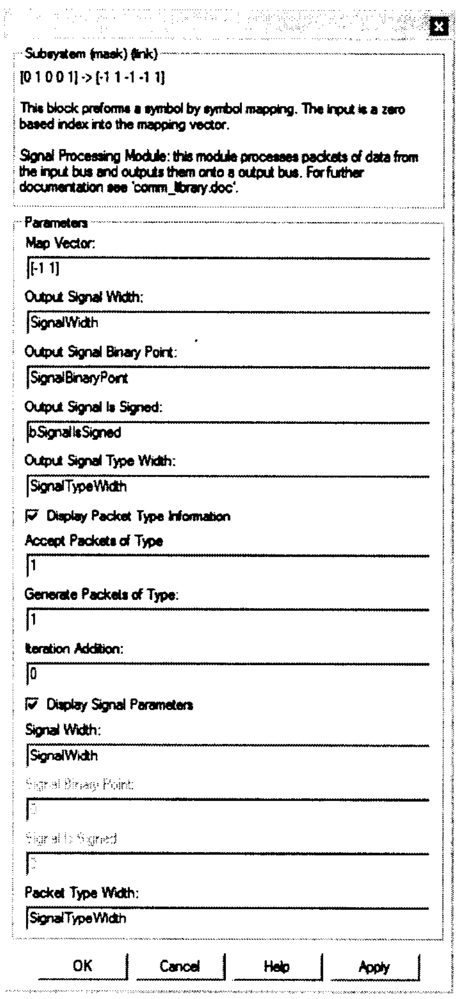


Figure 3.12: *Map Symbol* module



Subsystem (mask) (link)

[0 1 0 0 1] -> [-1 1 -1 -1 1]

This block preforms a symbol by symbol mapping. The input is a zero based index into the mapping vector.

Signal Processing Module: this module processes packets of data from the input bus and outputs them onto a output bus. For further documentation see 'comm_library.doc'.

Parameters

Map Vector: [-1 1]

Output Signal Width: SignalWidth

Output Signal Binary Point: SignalBinaryPoint

Output Signal Is Signed: bSignalIsSigned

Output Signal Type Width: SignalTypeWidth

[✓] Display Packet Type Information

Accept Packets of Type 1

Generate Packets of Type: 1

Iteration Addition: 0

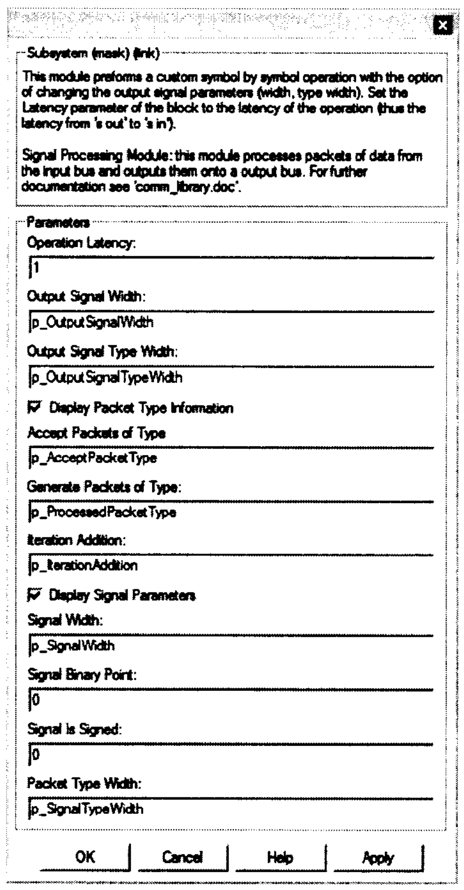
[✓] Display Signal Parameters

Signal Width: SignalWidth

Signal Binary Point: 0

Signal Is Signed [illegible]

Packet Type Width: SignalTypeWidth



Subsystem (mask) (link)

This module preforms a custom symbol by symbol operation with the option of changing the output signal parameters (width, type width). Set the Latency parameter of the block to the latency of the operation (thus the latency from 's out' to 's in').

Signal Processing Module: this module processes packets of data from the input bus and outputs them onto a output bus. For further documentation see 'comm_library.doc'.

Parameters

Operation Latency: 1

Output Signal Width: p_OutputSignalWidth

Output Signal Type Width: p_OutputSignalTypeWidth

[✓] Display Packet Type Information

Accept Packets of Type p_AcceptPacketType

Generate Packets of Type: p_ProcessedPacketType

Iteration Addition: p_IterationAddition

[✓] Display Signal Parameters

Signal Width: p_SignalWidth

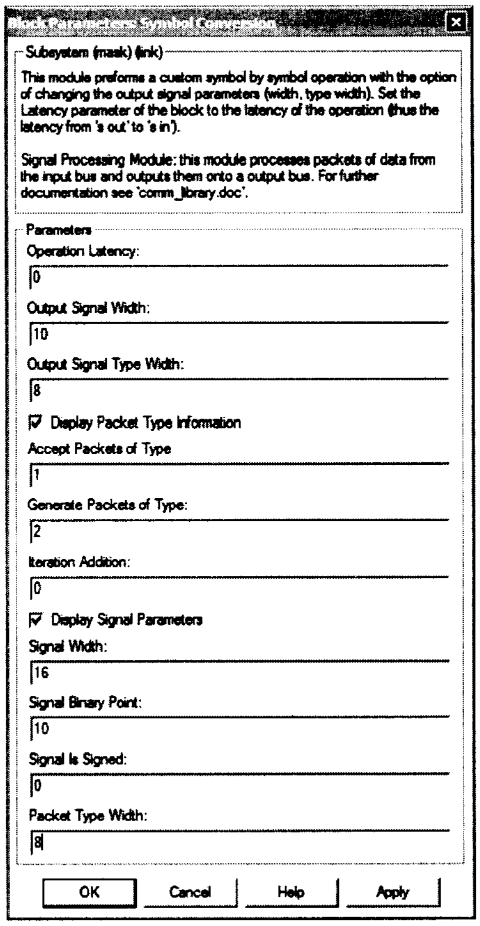
Signal Binary Point: 0

Signal Is Signed: 0

Packet Type Width: p_SignalTypeWidth

Figure 3.13: Passing parameters to an inner module

The *Fetch Packet* and *Packet Composer* blocks have no memory. Thus the *Fetch Packet* block outputs a packet to its output as it comes and the *Packet Composer* needs to be fed with packet data after we signal it to send a packet. Both these blocks can be replaced by blocks with memory. The *Store Packet* block can replace the *Fetch Packet* block. Since it stores the incoming packet in memory, where the packet can be accessed using the memory interface. For more information on the mentioned blocks, please see the block library documentation in Appendix B. The *Packet Composer* can be replaced by *Send Packet* block, which has memory and allows us to write a packet into it. When we are satisfied with the content of the memory we just signal it to send it as a packet. The *FIR Filter* module depicted in Figure 3.11 is illustrating the use of these modules.



Block Parameters: Symbol Conversion

Subsystem (mask) (link)

This module preforms a custom symbol by symbol operation with the option of changing the output signal parameters (width, type width). Set the Latency parameter of the block to the latency of the operation (thus the latency from 's out' to 's in').

Signal Processing Module: this module processes packets of data from the input bus and outputs them onto a output bus. For further documentation see 'comm_library.doc'.

Parameters

Operation Latency: 0

Output Signal Width: 10

Output Signal Type Width: 8

[✓] Display Packet Type Information

Accept Packets of Type: 1

Generate Packets of Type: 2

Iteration Addition: 0

[✓] Display Signal Parameters

Signal Width: 16

Signal Binary Point: 10

Signal Is Signed: 0

Packet Type Width: 8

Figure 3.14: The *mask* of the *Symbol Conversion* module

Alternatively, a module can be implemented by extending the functionality of an existing module. A good example is the *Map Symbol* module which implements the mapping of symbols *0* through *N* to any given symbols. This module is implemented using the *Symbol Conversion* module, as seen in the schematics in Figure 3.12.

In this case, parameters passed to the outer module inside its *mask* have to be passed to the inside module. This is illustrated in Figure 3.13, where it can be seen that parameters like *Signal Binary Point* and *Signal Is Signed* are required to be zero. Consequently, these parameters are set to zero in the mask of the inner module and disabled in the outer mask.

3.2.2 Masking a Module

Several customizations have to be applied to properly *mask* a module block. Please see the Appendix B for rules on how to mask a module. The mask of a module generally consists of three parts:

- Module specific parameters,
- Information on what packets should be processed and how,
- Specifics of the input signal (e.g., width, binary point).

If the parameters of the output signal of a module differ from the parameters of the input signal, the module specific parameters will also include the parameters of the output signal. This is the case in Figure 3.14, where the 'Output Signal Width' and 'Output Signal Type Width' define parameters of the output signal of the Symbol Conversion module. The 'Operation Latency' is an additional module specific parameter.

The next part of the mask dialog are the packet type information parameters. These parameters specify which packets should be processed and give the user the option to modify the 'Iteration' parameter within the packet. Finally, the last batch of parameters defines the expected parameters of the input signal. This includes:

- 'Signal Width', the bit width of the signal,
- 'Signal Binary Point' and 'Signal Is Signed', which define the fixed point interpretation of the signal,
- 'Packet Type Width', the bit with of the 'Packet Type' parameter within packets on the input.

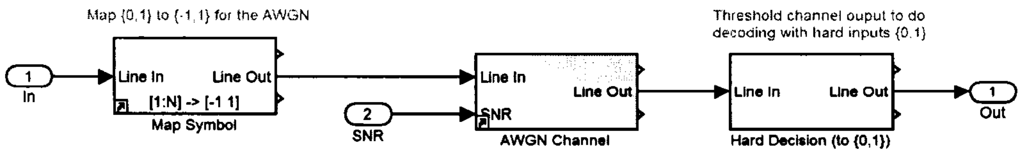


Figure 3.15: Implementing a BSC channel using AWGN channel

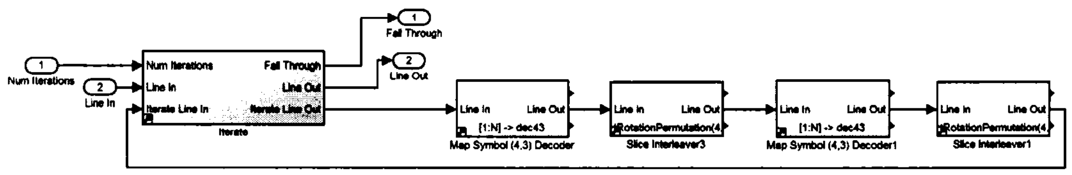


Figure 3.16: Iteration loop

3.3 Examples

One example of a system using our modular architecture is depicted in Figure 3.3. In this section, we provide additional examples that illustrate the following advantages of our modular design:

- Ease of implementation
- Clean and clear schematic representation

The first example, in Figure 3.15, illustrates the packet principle that enables easy linking of functional modules. In this example, three modules are linked and each of these modules implements a simple operation on the packet. First, a packet is translated from {0, 1} binary alphabet to {-1, 1} alphabet of binary antipodal symbols. Next, the *AWGN Channel* module accepts packets in this format and adds Gaussian noise to each symbol using the Gaussian random generator provided by *System Generator*. Finally, the values of the packet symbols are tested against zero threshold and mapped to either 0 or 1. This implements the functionality of a *Binary Symmetric Channel (BSC)*. We could have implemented the *BSC* directly, while using fewer resources, nevertheless the schematics in Figure 3.15 is a good example of linking functional processing modules.

The next example illustrates the implementation of an iterative loop. The *Iterate* module in Figure 3.16 inserts a packet into the loop through the four functional modules. While processing a packet, these modules increate the *Iteration* parameter stored within the packet. According to this parameter, the *Iterate* module then either reinserts the packet into the loop or outputs it on the *Line Out* output. The number of iterations can be controlled by an input of the *Iterate* module.

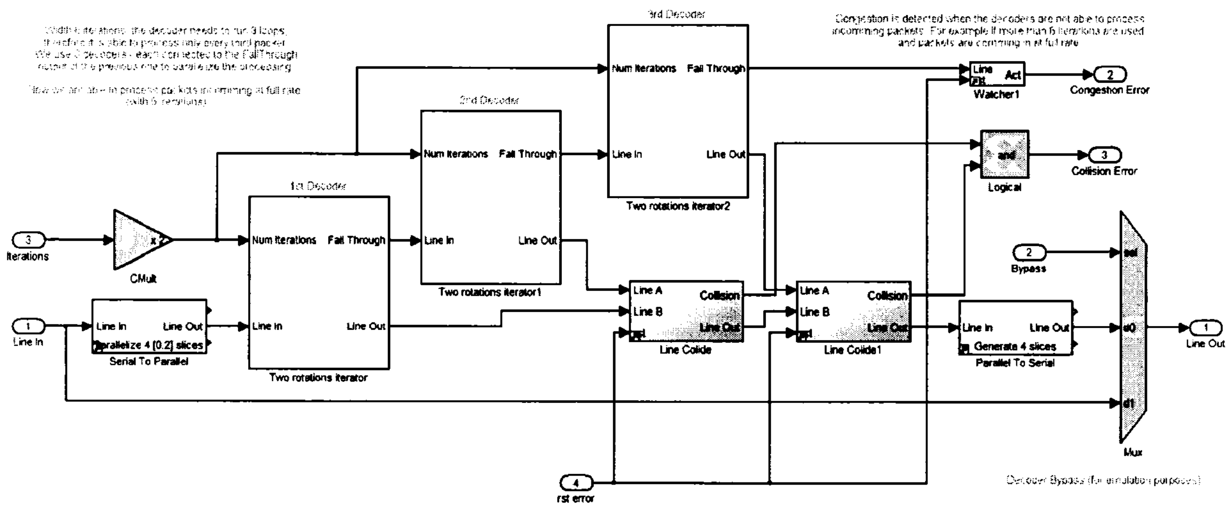


Figure 3.17: Parallelizing designs using the *FallThrough* output

The loop in Figure 3.16 implements two decoding iterations of the *Turbo Product Code* described later in Chapter 1.

The last example in this section, depicted in Figure 3.17, demonstrates the use of the *FallThrough* output which is part of all our modules. Packets that cannot be processed by a module (e.g., the module is busy) are routed to the *FallThrough* output. In Figure 3.17, the processing of the *Two Rotations Iterator*, which is the loop from previous example in Figure 3.16, is parallelized by chaining the *FallThrough* line through multiple copies of this module. This simple construct enables the decoder to process packets one after another, even though the actual decoder modules can accept packets only at a rate of $1/3$.

3.4 Chapter Summary

In this chapter, we have outlined the modular concepts employed in the development of our prototyping platform. We have defined a packet interface used for communication between modules. Furthermore, we have reviewed the construction of modules compliant with the modular and packet architectures. Finally, we have given an example of a system, which takes advantage of the proposed packet protocol.

Chapter 4

Developed Generate Block

In this chapter, we present the design of a *Generate* block for *System Generator.* This block allows automatic creation of regular logic cells and interconnects based on external parameters. We describe the design methodology using our Generate block which enables design time customizability of structures for various arrangements (linear arrays, matrices, trellises or trees). The Generate block effectively parallelizes operations, such as matrix multiplication, computation of a stage of the Viterbi or BCJR algorithm employed in decoding of a block or convolutional code. This block allows running FEC decoders at the required high speeds, while allowing these algorithms to be fully design time customizable. The work in this chapter was originally presented in part in [17].

4.1 Generate Block Introduction

In hardware implementations of DSP algorithms, we often employ designs that consist of arrays of regular cells. Examples of such algorithms are parallel matrix multiplication or FIR filtering. Implementations of these algorithms instantiate a linear array of cells with identical hardware logic in order to parallelize inner product computation. Nevertheless, if we want to implement modules that will perform the abovementioned operations, all parameters of the modules (e.g., FIR filter coefficients, filter constraint length) should be customizable. Consequently, we will need to programmatically create the arrays of regular cells depending on the parameters.

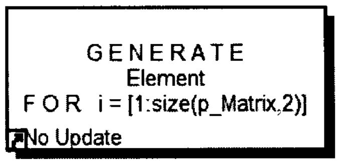


Figure 4.1: The Generate Block

In *VHDL*, we could create the regular structures using a generate statement or produce the VHDL source code using other programming language, such as *C* or *C++*. Unfortunately, a generate statement is not available in *System Generator*, so to implement automatic generation of regular structures, we will need to automatically create *System Generator* schematics. This can be done using *Matlab* functions for *Simulink* schematics manipulation, such as add_block, set_param and get_param. Using these functions in scripts, we can design a block (the *Generate Block* for *System Generator*) that will automatically create copies of a specified cell. This block also allows us to customize the interconnection of the created cells and thus create custom hardware directly from a specification in a *Matlab* variable.

4.2 Generate Block Implementation

4.2.1 Automatic Cell Replication

The *Generate* block allows automatically generating schematic designs that have regular structure and it is the *System Generator* analog of the VHDL generate statement. The block simply programmatically replicates a subsystem that is a part of the design specifying a regular cell. The *Generate* block takes two essential parameters:

- A reference to a subsystem containing the design that is to be replicated,
- A vector of values passed to the replicas allowing these to be parameterized.

Figure 4.2 gives us an example of the use of the *Generate* block with Figure 4.2(a) showing the integration of the *Generate* block into the design and Figure 4.2(b) depicting an example of the contents of the generated cell. Figure 4.3 displays the mask parameters of the *Generate* block. The front-end of the

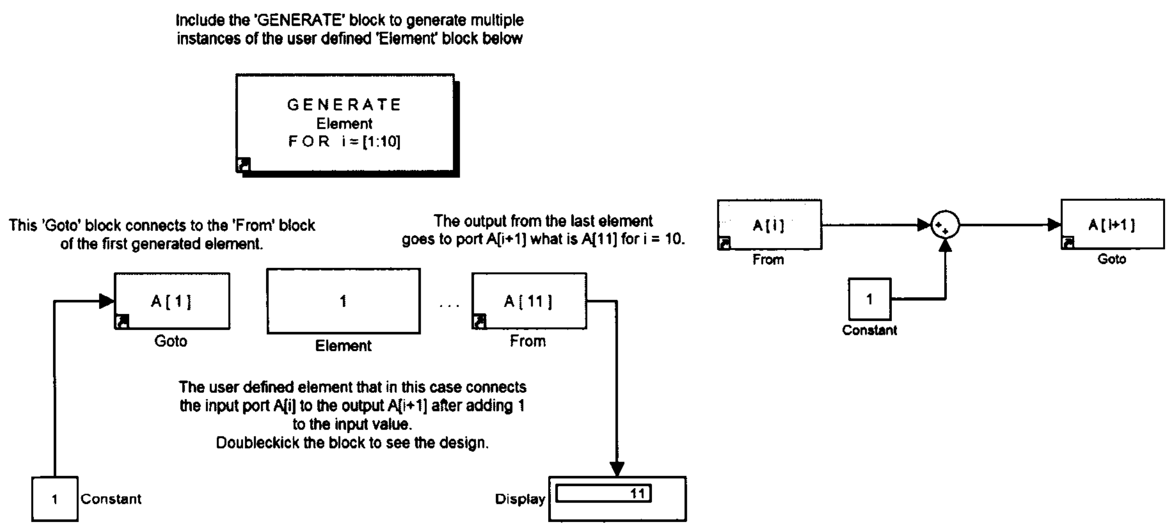
Generate block in Figure 4.2(a) displays a logical description of the idea of the design:

Generate 'Element' for i = [1:10],

where, the subsystem Element (note that the subsystem's name is specified in the mask) is replicated 10 times for the control variable i set to 1, 2, 3 … 10. The control variable is passed to the individual replicas and can be used to customize their behaviour. Figure 4.2(b) illustrates that the *From* and *Goto* blocks are dependent on the control variable i.

4.2.2 Interconnection of Generated Cells

On top of selecting the cell to replicate, the user of the *Generate* block has to specify how to interconnect the replicas. A cell can be connected to multiple other cells by a number of inputs and outputs. We have implemented special port blocks that enable us to specify inter-cell connections. In fact, these port blocks are nothing more than regular labelled ports that can be parameterized by the control variable of the *Generate* block.


Include the 'GENERATE' block to generate multiple instances of the user defined 'Element' block below
GENERATE
Element
FOR i=[1:10]
This 'Goto' block connects to the 'From' block of the first generated element.
The output from the last element goes to port A[i+1] what is A[11] for i = 10.
A[1]
Goto
1
Element
...
A[11]
From
The user defined element that in this case connects the input port A[i] to the output A[i+1] after adding 1 to the input value.
Doubleckick the block to see the design.
1
Constant
Display
11
A[i]
From
1
Constant
A[i+1]
Goto

(a) Generating 10 instances of the subsystem Element

(b) The content of the subsystem Element

Figure 4.2: Generic design illustrating the use of the developed *Generate* block

Each *From* block bridges a signal from a *Goto* block with the same label. Thus a *From* block labelled Superlink would output the value of a signal connected to a *Goto* block with the label Superlink somewhere else in the design. In our case, a *From* block labelled A[i] in a cell generated for i = 3 would connect to a *Goto* block labelled A[i+1] in a cell generated for i = 2, because in both cases the labels become A[3].

Note that instantiating the cell in Figure 4.2(b) for i from 1 through 10 will link these into a linear chain connecting each instance of a cell to the previous and the next one. The only instantiated ports that will not be connected are A[1] and A[11] ports of the first and the last cell, respectively. We simply connect these ports by manually instantiating ports with the appropriate labels in the surrounding design, as seen in Figure 4.2(a). Ports A[1] and A[11] interface with ports inside the generated design. Thus the example instantiates 10 cells, where each one takes the input, adds one to it and forwards the signal to the next cell. Therefore the overall generated design in this example will just add 10 to the input signal (we see that the Display block indeed shows 1+10 = 11).

Note that all these ports, A[1] through A[11], can be further parameterized with other external variables while using any *Matlab* expressions. With the power of *Matlab* expressions, we are able to create front-end customizable, automatically generated regular cells with complex interconnect structures. We further discuss the construction of such structures in subsection 4.3.3.

4.2.3 Contextual Behaviour of Several Generate Blocks

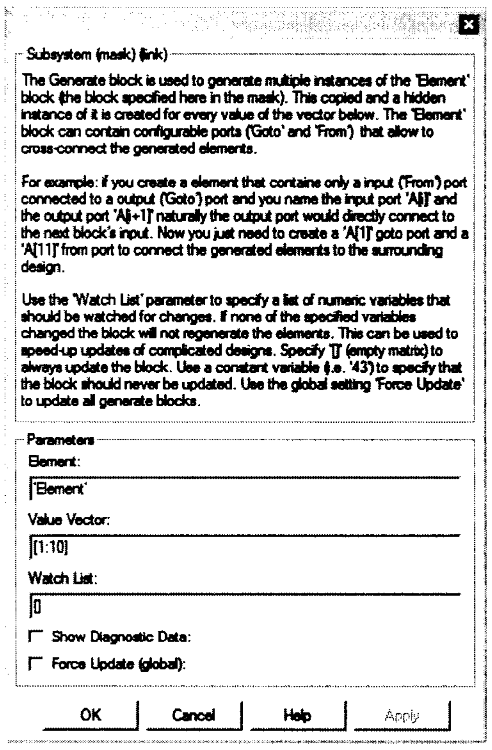
To facilitate usage of multiple *Generate* blocks in a design, we define an identity for each such block. Using the identity, the port blocks will be associated with a specific *Generate* block and will work in a common context, while not interfacing with port blocks of other *Generate* blocks. The following rules govern the context of port blocks:

- A port block placed in the same subsystem with a *Generate* block will work in the context of this block, interfacing with the generated design inside the cells. Such port block will display yellow.

- A port block inside a generated cell will work in the context of the *Generate* block the generated the cell. Such port block will display red.

Additional rules are applied to resolve conflicts resulting from the above rules:

- In case of nested blocks, both of the above cases might be true. Consequently, the context of a port block is set by an 'Ignore Scope' option inside this block. The default behaviour is that the port block is associated with the *Generate* block included in the same subsystem. Nevertheless, if the 'Ignore Scope' option is set to be *on*, the *Generate* block will be ignored and the block will be associated with a higher level *Generate* block.
- Two *Generate* blocks that are defined in the same subsystem will share contexts, thus allowing design of regular structures with two or more structural modes. In this case, it is desirable that structures generated by different *Generate* blocks interface.


Subsystem (mask) (link)

The Generate block is used to generate multiple instances of the 'Element' block (the block specified here in the mask). This copied and a hidden instance of it is created for every value of the vector below. The 'Element' block can contain configurable ports ('Goto' and 'From') that allow to cross-connect the generated elements.

For example: if you create a element that contains only a input ('From') port connected to a output ('Goto') port and you name the input port 'A[i]' and the output port 'A[i+1]' naturally the output port would directly connect to the next block's input. Now you just need to create a 'A[1]' goto port and a 'A[11]' from port to connect the generated elements to the surrounding design.

Use the 'Watch List' parameter to specify a list of numeric variables that should be watched for changes. If none of the specified variables changed the block will not regenerate the elements. This can be used to speed-up updates of complicated designs. Specify '[]' (empty matrix) to always update the block. Use a constant variable (i.e. '43') to specify that the block should never be updated. Use the global setting 'Force Update' to update all generate blocks.

Parameters

Element: 'Element'

Value Vector: [1:10]

Watch List: []

☐ Show Diagnostic Data:

☐ Force Update (global):

Figure 4.3: The parameter dialog of the *Generate* block

4.2.4 Implementation of the Generate Block

The realization of the *Generate* block consists of the implementation of the following three parts:

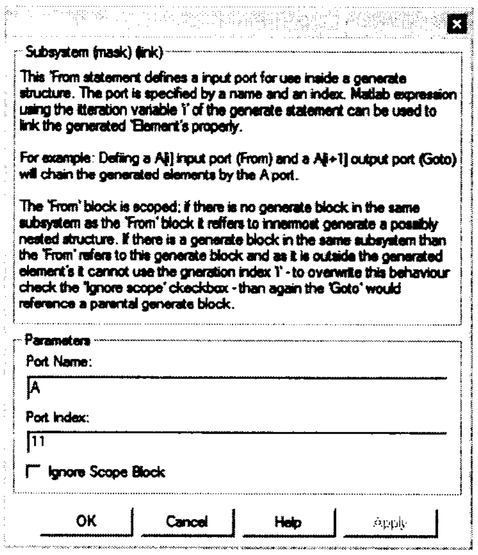
- Integration of the *Generate* block into the *Simulink* environment enabling the user to specify replication parameters and allow the subsystem to replicate.
- Realizations of *From* and *Goto* port blocks that enable the user to create custom interconnect among the generated cells.
- Programmatic replication of the specified subsystem that should occur automatically before a simulation or translation to hardware is executed. This includes passing of parameters to each generated cell, specifying its identity and context (this enables parameterization of the cell's blocks which is required to enable the creation of custom built interconnect structures).

The first part is realized by including the *Generate* block within a *Simulink* design that requires the automatic generation of regular structures so that, this block will store the replication parameters. The parameters can be modified via the block's properties dialog box (in *Simulink* this dialog box is called a mask), as shown in Figure 4.3. The *Generate* block's parameters include a by-name reference of the subsystem that is to be replicated, a vector of values (each replica will be associated with one value from this vector) and parameters controlling the update process of the generated schematics.

The second part of the implementation of the *Generate* block is the realization of the *From* and *Goto* blocks. *Simulink* includes generic *From* and *Goto* blocks that enable routing of a signal without actually connecting blocks by a line. *From* and *Goto* blocks are considered connected if they have the same label (a *From* block labelled A is connected to a *Goto* block labelled A). Consequently, the *From* and *Goto* blocks that we use with our *Generate* block are simply instances of the generic *Simulink* blocks, where the labels are further parameterized by GenerateID and i, thus creating a block labelled A_GenerateID_f(i). An initialization script included in these blocks assures that the labels are updated properly. Figure 4.4 displays the parameter dialog box of a port block that enables us to specify the port label as well as the function that specifies how to evaluate the port index f(i).

The last part of the implementation of the *Generate* block is realized as a *Matlab* script. The *Generate* block uses the *Matlab* functions add_block, delete_block, set_param and get_param, which enable creation and modification of *Simulink* schematics, in order to automatically replicate the specified subsystem. This script is included in the *Generate* block's initialization code that is executed before each simulation. This assures that the created replicas will be up-to-date and identical with the template subsystem.

The update process can take considerable time and if no modifications were made to the template subsystem, it is not necessary. Therefore, we give the user an option to disable the automatic update. Another option is to specify a list of parameters that should be watched and the generated schematics is updated whenever some of these parameters change. Moreover, the script parameterizes the generated replicas with two further parameters: GenerateID (the identity of the generate module that created this replica) and i (the *index* of this replica – a value from the vector specified in the *Generate* block). These parameters are then used to properly parameterize the *From* and *Goto* blocks.


Subsystem (mask) (link)

This 'From statement defines a input port for use inside a generate structure. The port is specified by a name and an index. Matlab expression using the iteration variable 'i' of the generate statement can be used to link the generated 'Element's properly.

For example: Defiing a A[i] input port (From) and a A[i+1] output port (Goto) will chain the generated elements by the A port.

The 'From' block is scoped; if there is no generate block in the same subsystem as the 'From' block it reffers to innermost generate a possibly nested structure. If there is a generate block in the same subsystem than the 'From' refers to this generate block and as it is outside the generated element's it cannot use the gneration index 'i' - to overwrite this behaviour check the 'Ignore scope' ckeckbox - than again the 'Goto' would reference a parental generate block.

Parameters

Port Name:

A

Port Index:

11

☐ Ignore Scope Block

Figure 4.4: The parameter dialog of a *From* block

4.3 Design with the Implemented Generate Block

In this section, we will present several examples of the use of the *Generate* block in different scenarios. These examples include the use of multiple interfacing or non-interfacing *Generate* blocks, nesting of *Generate* blocks, structuring the generated cells in a tree or a trellis alignment and the implementation of a matrix multiplication module.

4.3.1 Multiple Generate Blocks

The generate block can be used to replicate cells with identical structure. Nevertheless, occasionally we will need to generate structures that contain multiple types of cells. We can achieve this by using multiple *Generate* blocks, as discussed in subsection 4.2.3. If we include these blocks in the same subsystem, the port blocks of the generated cells will interface and an interconnect structure between various cells can be created. This is the case in Example A in Table 4.1, where we generate an array of cells with the first, third, fourth, fifth, seventh and tenth cells based on Element A and the second, sixth, eighth and ninth based on Element B. Matlab expressions can be used for the *i*-value vectors, when generating more complex structures.

On the other hand, if we use a *Generate* blocks to implement several features in one design, we will place the *Generate* blocks in separate subsystems. This way, the generate blocks will not have the same context and there will be no interference between these two blocks.

4.3.2 Nesting of *Generate* Blocks

Generate blocks can be used also inside cells generated by another *Generate* block and this functionality is called *nesting*. Nesting is useful to create arrays of regular cells that extend in multiple dimensions or when employing features that require a generate statement (such as parallel matrix multiplication) inside a generated cell. Nesting of *Generate* modules can be used transparently. In some instances of nesting, it might be unclear whether a port block belongs to the inner or the outer *Generate* block. A special option is included in the port block that enables the user to resolve such ambiguities.

4.3.3 Structured Interconnect

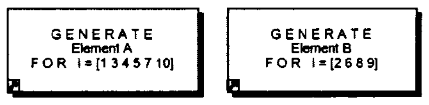
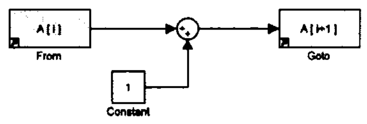
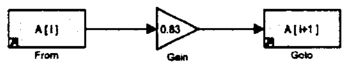
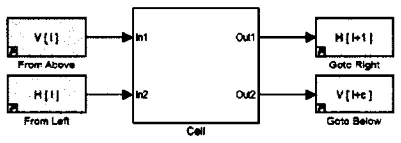
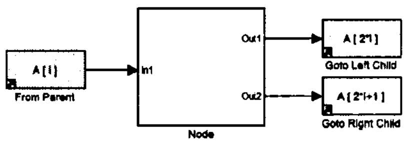
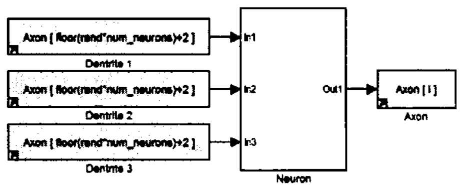
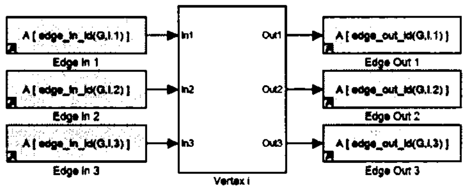
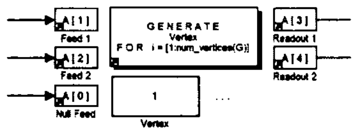
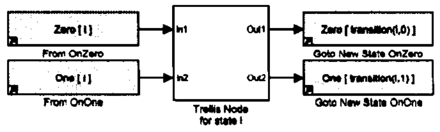
In the subsections 4.2.1 and 4.3.1, we have used the *Generate* block to create linear arrays of cells. Nevertheless, different arrangements of cells with different interconnect structures might be desired in practice:

- A rectangular grid of four-way connected processing cells (or other grid types) might be desired when solving partial differential equations in two dimensions as in [25] (Example B in Table 4.1),
- Cells might need to be arranged into a tree structure [31] for effective parallelization or aggregation of signals. For example, to automatically pipeline an adder of a big number of signals (Example C in Table 4.1).
- A randomized interconnect of a large number of cells might be desired when doing simulations of a neural network, as in [28] (Example D in Table 4.1).
- Arrangements into a fixed graph might be useful for simulation of computer networks or logistic problems [31] (Example E in Table 4.1).
- Trellis arrangements are common in many communication algorithms (e.g., BCJR [3], SOVA [15]) (Example F in Table 4.1).

All these arrangements can be implemented using *Matlab* expressions in the port indices. We have used the port labels A[i] and A[i+1] to link cells into a linear array. With other expressions, we can achieve grid, tree, trellis and other arrangements. In Example B, we use the label A[i+C] to connect cells across rows in a grid structure. Example C uses the port labels A[i], A[2*i] and A[2*i+1] to arrange cells into a tree structure.

In fact, we can use any expression, build-in *Matlab* function or user defined function in the evaluation of the port index. This gives us tremendous power to use highly sophisticated definition of the interconnect structure. In Example E, we have a graph specified as a *Matlab* variable G. We define two functions, edge_in_id(G,v,j) and edge_out_id(G,v,j), that will return a numeric identifier of the j-th edge entering and leaving vertex v (assuming edges incident with a vertex are

Table 4.1: Using the *Generate* block to create cell structures of various arrangements

Example A: Generating an linear array consisting of cells of two different types	
Two *Generate* blocks placed in the same subsystem will generate cells with interfacing port blocks. This example will generate an array of **A**- and **B**-type cells. The value vectors of the *Generate* blocks determine the cell arrangement.	
Content of Element A	
Content of Element B	
Example B: Generating a grid of cells	
Generating this cell for **i = [1:c*r]** will create a rectangular grid with c columns and r rows. Ports labelled V connect the cells vertically and ports labelled H horizontally. In this configuration, communication between cells is only possible from left to right and from top to bottom.	
Example C: Generating cells arranged in a tree	
Generating for **i = [1:n]** will create a tree where we can use **A[0]** to feed the root and **A[n+1]** to **A[2*n+1]** to retrieve results from the nodes. This arrangement can be useful to generate a structure that will feed the grid from example B with data if the number of columns in that example is unknown.	
Example D: Creating randomly interconnected cells	
Generating these cells creates an array of randomly interconnected neurons. This design will create a different interconnect structure each time it is compiled. In practice, we would probably use a pre-generated random vector to define the interconnect. On top of the generated neurons, we can manually instantiate some neurons with extra inputs and outputs to excite and monitor the network.	
Example E: Generating cells connected into a structure given by a graph	
We can number the edges of a graph G and create custom *Matlab* functions **edge_in_id(G,v,j)** and **edge_out_id(G,v,j)** that will return the number of the j-th edge entering or leaving the vertex v in graph G. Then we can use this cell to generate cells connected by signals given by the graph G (if there are no more than three edges entering or leaving any vertex in G).	
To interface with the design, we can "reserve" the edge numbers 1, 2, 3, 4 for edges feeding and reading-out data. A null-feed source has to be created to feed the ports of vertices that have less than three incoming edges in G.	
Example F: Creating a trellis stage	
A trellis stage can be automatically created by generating the cell on the right for all states i. The function **transition(state,symbol)** specifies the state transition on the given symbol.	

ordered) in graph G. Now assuming that there are no more than three edges entering or leaving any vertex in G, we can generate cells interconnected in a structure defined by graph G using:

Generate 'Vertex' for i = [1:num_vertices(G)]

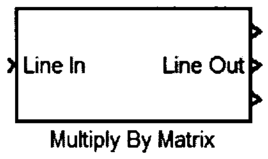
to generate vertices that would contain input ports A[edge_in_id(G,i,1)], A[edge_in_id(G,i,2)], A[edge_in_id(G,i,2)] and output ports A[edge_out_id(G,i,1)], A[edge_out_id(G,i,2)], A[edge_out_id(G,I,3)].

In Example D, we have create a cell called Neuron with an output port Axon[i] and three input ports Axon[floor(rand*num_neurons)+1], Axon[floor(rand*num_neurons)+1] and Axon[floor(rand*num_neurons)+1]. Generating these cells for i = [1:num_neurons] will create a neural network with random connection between neurons. In the last example in Table 4.1 (Example F), we have demonstrated the automatic generation of a stage of a trellis graph.

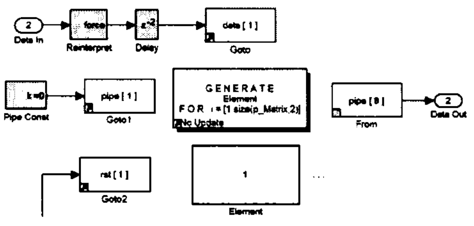
4.4 An Example Using the *Generate* Block

We conclude this section with an example of the full implementation of a matrix multiplication module, as seen in Figure 4.5. This implementation uses the *Generate* block to parallelize its operation. The module multiplies a row vector given at the input as a sequence of symbols by a given matrix from right and outputs the resulting row vector sequentially. One cell computes the inner product of the input with one column of the matrix. Using the *Generate* block, such a cell is dynamically generated for each column of the given matrix and the outputs of these cells are then merged together using a *data-pipe* that connects the cells into a linear array.

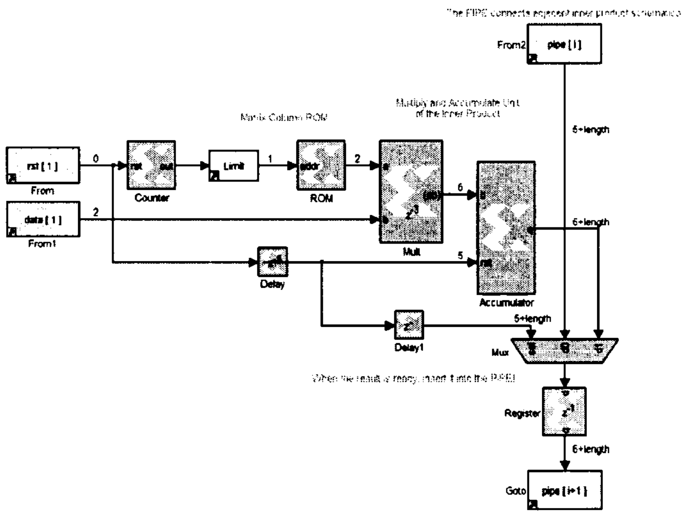
The design of our matrix multiplication module is simple. All cells compute the inner product in the same clock cycle. In this clock cycle, the control signal of the *Mux* multiplexer block in Figure 4.5(b) is asserted and therefore all the *Register* blocks within the cells will be loaded with this result. In the following clock cycles, the control signal of the *Mux* will be low and therefore the *Register* blocks will be connected in a row. We call this row a *data-pipe*, because the data in these registers will shift by one cell in each clock cycle. Data at the last *Register* can be used to compose the output packet.



(a) The matrix multiplication module



(b) Connecting the generated cells



(c) A single cell of the matrix multiplication

Figure 4.5: Matrix multiplication module

This design employing the *Generate* block is used in the *Matrix Multiplication* module, which is included in the communication library described in Chapter 5. The parallelization enables the module to process packets one after another.

This matrix multiplication is used in the implementation of a fully parallel encoder and decoder for linear block error-control codes. Using the *Generate* block, it was possible to design the encoder and decoder without the prior knowledge of the length of the codeword or the generator matrix. The generator matrix is specified via the front end interface of the modules, thus enabling easy customization of the modules. Using a specific generator matrix, we can use the modules to encode and decode the Hamming (7, 4) code, as seen in Chapter 6.

4.5 Chapter Summary

We have designed a methodology that allows automatic creation of regular logic cells for FPGA hardware design within *System Generator.* We have implemented this methodology in the form of a *Generate* block. The use of the *Generate* block was demonstrated in detail on the implementation of a matrix multiplication module. In Chapter 6, we will describe how we successfully used the block to implement a Hamming (7, 4) encoder and decoder which we then employed in our optical network CDMA testbed.

Chapter 5

Implemented Prototyping Platform

This chapter describes the libraries, tools and schemes, which form our rapid system prototyping platform. Development process using this platform, consists of three steps:

- Algorithm design and development
- Testing of the developed algorithms
- Performance tests of the developed scheme in simulated real world environment

We have constructed a set of tools, schemes and modules that provide support with each of these steps. First, a library of high-level functional modules (i.e., modules that operate on data) is provided to aid in the development of algorithms. Next, an additional library with debugging blocks enables testing and verification of the developed algorithms. Finally, schemes are provided to enable automated testing of the developed algorithms in a simulated environment using hardware co-simulation.

Currently, we have used AWGN noise generated by our library modules for simulation of the adverse effects of the environment which include multiuser interference, modulation imprecision and channel noise due to receiver electronics. The AWGN noise model is appropriate for regular-power optical CDMA applications. Nonetheless, in the future other noise models as well as direct simulation of the interfering signals can be considered.

The following sections describe the abovementioned libraries, tools and schemes.

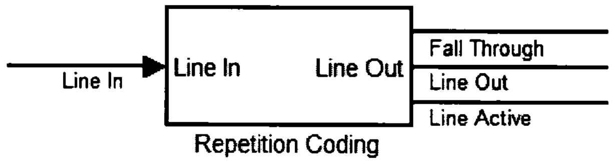


Figure 5.1: A functional communication system module

5.1 Implemented Functional Modules Library

The core of our rapid prototyping platform is a library of packet handling modules for communication systems, specifically optical OCDMA. The modules can be used in other signal processing applications, such as, Software Defined Radio [34], [31]. This library includes:

- Modules for packet data conversion and simple operations (e.g. converting symbols from x to y bits, taking symbols modulo b, mapping zero and one symbols to minus and plus one).
- Modules executing vector operations (e.g., multiply vector by matrix, interleave data packet, FIR filtering).
- Packet form editing modules (e.g., cut subsequence from packet, subsample a packet, code a packet using repetition-code, parallelize and serialize packet).
- Modules for communication systems simulations (e.g., random packet generation, AWGN channel simulation, error counting modules).
- Packet routing and other utility modules (e.g., iterate packet in a loop, input line aggregation modules, packet storing and retrieval modules, header information retrieval modules).

Additionally, some error control solutions, described in greater detail in Chapter 6, have been included in the library:

- Communication systems algorithms (e.g., FSM Encoder, Viterbi Decoder, Block Code Encoder, Syndrome Decoder)

All modules are optimized for high performance and have the same outward interface, depicted in Figure 5.1 and described in Chapter 3. Some of the modules are implemented using a special *Generate* block, described in Chapter 4, to allow

Table 5.1: Data conversion and simple operations modules

Module	Description
Symbol Operation	Executes a custom operation symbol by symbol (e.g., add 2)
Symbol Conversion	Changes the format of packet symbols (e.g., its bit width)
Map Symbol	Applies a custom mapping to packet symbols (e.g., $\{0,1,2\}\rightarrow\{-1,1,0\}$)
Hard Decision	Maps all negative symbols to *-1* and all positive to *+1*

automatic creation of arrays of regular structures for parallel processing. In section 0, the chapter also explains how this block is used to parallelize the processing in the *Matrix Multiplication* module. For details on module implementations and parameter/signal specifications as well as on the specifics of how to use the library modules in a design please refer to Appendix B.

5.1.1 Data Conversion and Simple Operations Modules

The elementary Packet/Data handling modules allow us to do elementary operations on data contained within a packet. These modules, such as *Symbol Operation* and *Symbol Conversion*, enable us to execute operations on all data symbols within a packet. Additional modules execute specific symbol operations. The overview of these modules is presented in Table 5.1 and detailed description of the modules is given in Appendix B.

Table 5.2: Vector operations and manipulation modules

Module	Description
Multiply By Matrix	Multiplies the packet interpreted as a row vector by a fixed matrix from right
Interleaver	Permutes the symbols in a packet according to a user provided permutation
FIR Filter	Convolves the packet with a given vector
Parallel FIR Filter	Executes multiple *FIR Filter* operation on the packet in parallel
Vector Operation	This module enables us to define a custom operation on two vectors
Vector Aggregate	Executes a custom operation on two vectors with output aggregation

5.1.2 Vector Operations and Manipulation Modules

These modules interpret the packets as data vectors and execute operations on these vectors. Modules accept packets of arbitrary length, dynamically adjusting the operation to the length of the packet. Nevertheless, usually the modules require the maximum packet length to be defined as a parameter. The overview of these modules is presented in Table 5.2. For detailed information on the modules, please refer to Appendix.

5.1.3 Packet Form Editing Modules

The packet-form editing modules enable us to alter the structure of a packet by cutting symbols out of a packet, repeating symbols or aggregating symbols into new symbols of increased bit width. The overview of these modules is presented in and the detailed information is provided in Appendix B.

5.1.4 Packet Routing and Other Utility Modules

This section of the library includes modules that enable us to route packets between modules, process packets in an iterative loop, store packets and load packets from memory. The overview of these modules is presented in Table 5.4. Detailed information is provided in Appendix B

Table 5.3: Packet-form editing modules

Module	Description
Cut From Packet	Enables to cut a continuous data block out of the middle of the packet
Serial To Parallel	Aggregates consecutive symbols into one symbol of increased bit width
Parallel To Serial	Slices a symbol and represents it in the output packet as multiple symbols
Repetition	Repeats each symbol in a packet a given number of times
Downsample	Puts every k-th input symbol of the packet into the output packet

Table 5.4: Packet routing and other utility modules

Module	Description
Iterate	Iterates a packet in a loop for a given number of times
Line Cat	Joins two packet transmission lines by synchronizing the packets on these lines and concatenating symbols from these packets
Line Collide	Joins two packet transmission lines, while simultaneous packets will collide
Store Packet	Stores a packet in memory
Send Packet	Creates a packet from memory
Header Spy	Extracts the header information from a packet
Packet Count	Counts the number of packets transmitted over a line

5.1.5 Communication Systems Algorithms

Several communication system algorithms are a part of the library. These include the encoder and decoder for block codes and a $(4,3)^4$ Turbo Product Code, a finite state machine encoder and a Viterbi decoder. Table 5.5 lists the implemented communication system modules; their detailed description is given in Appendix B.

Table 5.5: Communication system algorithms

Module	Description
FSM Encoder	Encodes a packet using a specific finite state machine (FSM)
Viterbi Decoder	Decodes packet encoded by either an *FSM Encoder* or an *FIR Filter*
Block Code Encoder	Encodes a packet using a block code
Syndrome Decoder	Decodes a linear block code using syndrome decoding algorithm
TPC Encoder	$(4,3)^4$ Turbo Product Code encoder
TPC Decoder	$(4,3)^4$ Turbo Product Code decoder

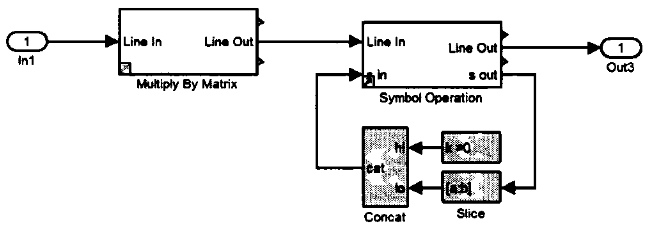


Figure 5.2: Implementation of the encoder for the Hamming (7,4) code

5.1.6 Building Error Control Modules Using the Functional Module Library

Most error control solutions can be straightforwardly implemented using known algorithms. These algorithms usually incorporate symbol manipulation, matrix multiplication and finite field algebra. The communication library provides support for all these operations.

In this section, we provide an example of the implementation of a Hamming (7,4) encoder and decoder. The Hamming code is a block code and thus the encoding process is simply a multiplication of the 4 bit input vector by the generator matrix over *GF(2)*, where our implementation in Figure 5.2 exactly expresses this idea.

In a syndrome decoder of the (7,4) Hamming code, we simply multiply the received vector by a parity check matrix *H* of this code, thus getting a syndrome. Then we use a mapping from the syndrome to identify the erroneous bits. Our implementation in Figure 5.3 copies this description:

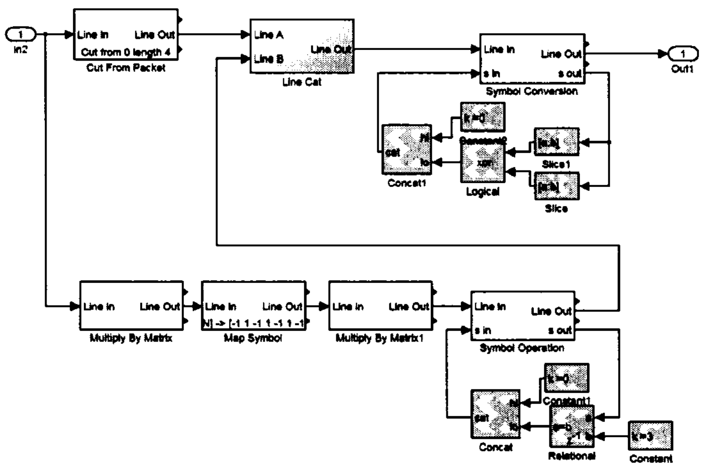


Figure 5.3: Implementation of the syndrome decoder for the Hamming (7,4) code

- Received vector is multiplied by the parity check matrix H to get the syndrome,
- The syndrome is multiplied by a matrix whose columns are matched to possible syndrome vectors, as explained in Chapter 6,
- A symbol operation module identifies the erroneous bits,
- The vector of erroneous bits is added (modulo 2) to the vector of original systematic bits.

The approach using a matrix with columns matched to possible syndrome vectors is convenient, with syndrome symbols available serially in a packet, as this approach does not require merging the syndrome symbols, using a look-up table and then serializing the corrections that need to be applied to the received vector. The Hamming (7,4) decoder is described in greater detail in Chapter 6.

5.2 Implemented Library of Debugging Blocks

We have developed a set of emulation and debugging modules that can be used to test the validity and performance of an implemented system. These modules include:

- Communication system simulation: random data generators, channel simulation, error counters,
- Data display and data analysis tools: display packets on a selected line, store packets for analysis in Matlab, automatically start/stop/pause logging of transmission on a data line and perform other diagnostics.

The modules in the first bullet can be used during *Simulink* simulation as well as for performance testing on the FPGA chip while the modules in the second bullet are only useful during *Simulink* simulation. Simulating the design within *Simulink* enables better observation of the behaviour of a design as signals values in each clock cycle can be stored in a *Matlab* variable and analyzed off-line. The FPGA will behave identically to the *Simulink* simulation, and therefore it is useful to debug designs within *Simulink*. This part of our library provides modules that make such debugging easier.

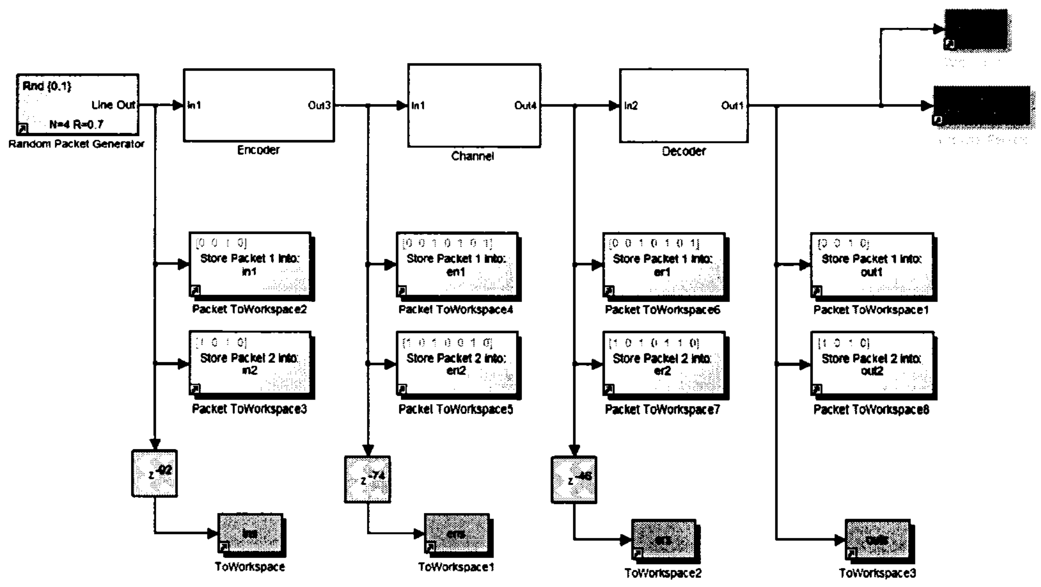


Figure 5.4: Testing a system in Simulink simulation

The system in Figure 5.4 integrates the Hamming encoder and decoder with a simulated channel and also includes modules that enable the developer to intercept and display packets transmitted between modules. In Figure 5.4, the first and second packets are displayed as they passed through the system in the respective rows of the *Packet ToWorkspace* modules, and with a very high resolution printer, you can observe that one error was actually corrected in the second packet.

Table 5.6: Communication system simulation modules

Module	Description
Random Packet Generator	Generates a packet containing random binary data
Vector Packet Generator	Creates packets specified by a parameter vector
Packet Generator	Creates packets filled with symbols from an input
AWGN Channel	Emulates the transmission over an AWGN channel
Count Errors on Random Packet	Counts the number of errors that occurred in transmission

5.2.1 Communication System Simulation Modules

The first set of modules facilitates evaluation of communication systems by providing random packet generators, channel simulation and error counting modules. By including these modules in a setup and translating this setup into an FPGA, bit-error-rate simulations can be executed at full hardware speed. Table 5.6 provides an overview of these modules. . For further details please refer to Appendix B.

5.2.2 Data Display and Analysis Blocks

The data display and analysis blocks enable the user to view the data transmitted over a line during a simulation and store these data into the *Matlab* workspace for later analysis. Functionality of the blocks is outlined in Table 5.7; Appendix B contains further details.

Table 5.7: Debug control and output modules

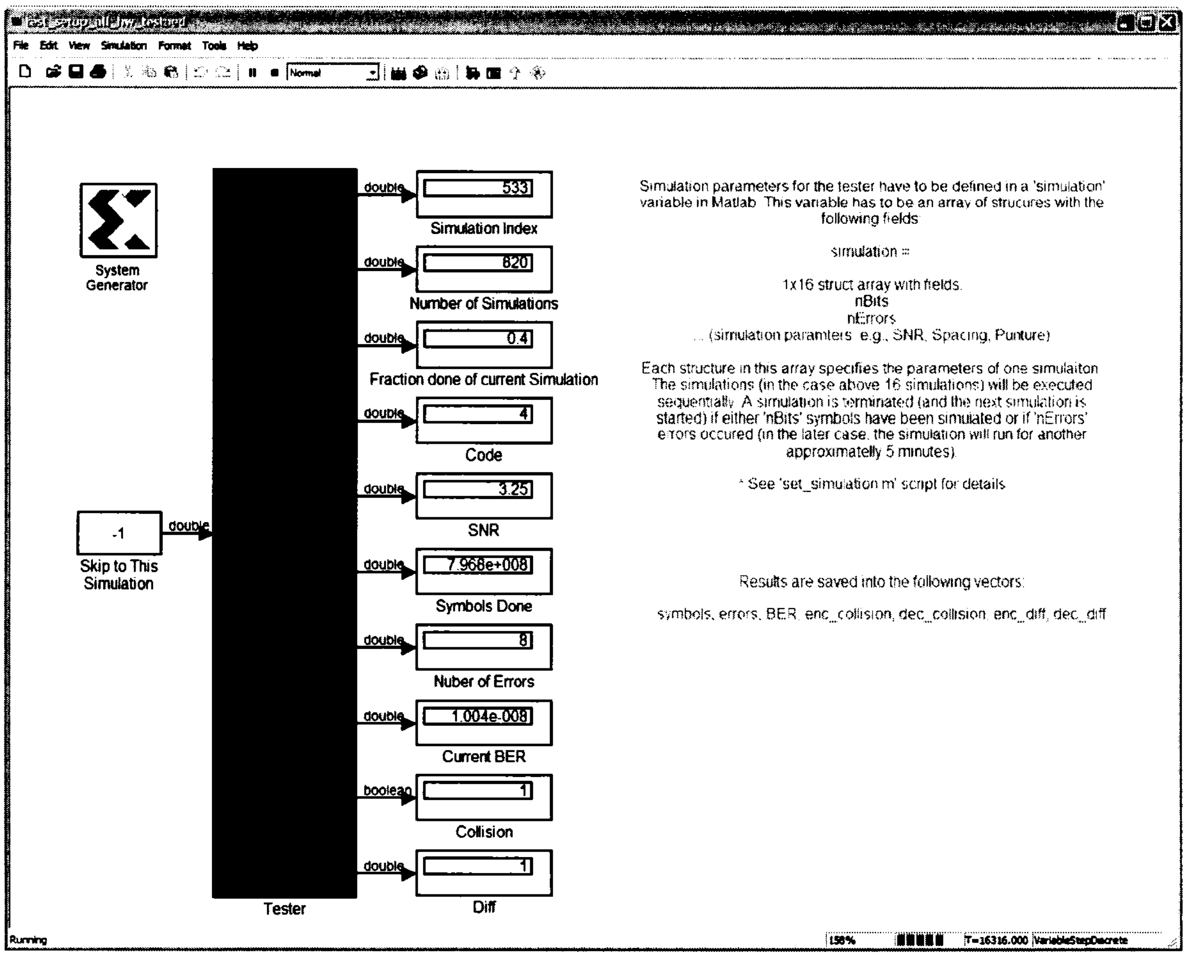
Module	Description
Start Logging	Logging of signals will start when the input of this block becomes nonzero
Start Immediately	Start logging signals immediately after the simulation begins
Stop Simulation	Stops the simulation when the input of the block becomes nonzero
Stop After Packets	Stops the simulation after *n* packets have been received by the block
Pause After Packets	Pauses the simulation after *n* packets have been transmitted
To Workspace	Stores all symbols on a line into a Matlab variable
Packet To Workspace	Logs memory write operations for later analysis in Matlab
Memory To Workspace	Stores the given packet transmitted over a line into a Matlab variable
Line To Workspace	Stores all packets on a line into a structured Matlab variable
View Index	Enables to display the index of a packet currently transmitted over a line

5.3 Automated Testbed: Blocks and Schemes for Hardware Co-Simulation

We have implemented an automated testbed for performance simulation. In this setup, *System Generator* enables to run an FPGA at full speed while embedding it into a *Simulink* simulation running on the host computer. Figure 5.5 shows the front end of a performance testbed used to test the performance of a communication system based on a turbo product code. Figure 5.6 and Figure 5.7 shows a more detailed view of the implementation of this testbed (the **Tester** block in Figure 5.5 contains the design in Figure 5.6, where in turn, the **Subsystem** block contains the design in Figure 5.7). The testbed enables to automatically run a batch of simulations on the FPGA chip, while the parameters for each simulation are specified in a *Matlab*, using a variable.

This testbed uses hardware co-simulation to integrate flexibility of executing control in *Simulink* and the high performance of running the actual algorithm on FPGA hardware. The communication between the hardware and *Simulink* is realized using a register transfer protocol (via a JTAG or a USB connection) as described in Section 2.2.5. We can use the *Simulink* simulation to control the FPGA hardware and to read and store results. While Figure 5.7 depicts the *Simulink* schematic that controls the execution of a sequence of simulations, Figure 5.7 shows the use of look-up tables to retrieve the parameters of each simulation and drive the input ports of the FPGA. Parameters of each simulation are stored in a *Matlab* structure and all simulations are run automatically one after another. In the case of a simulation for the Turbo Product Code, described in Section 6.3, simulation parameters would be packet spacing, SNR, bypass AWGN, number of iterations and number of punctured bits for testing purposes. The parameters of a simulation also specify stopping conditions, e.g., a simulation might finish after a certain number of symbols or a certain number of errors.

To run a series of simulations, we simply create an array of simulation-parameter structures and execute a *Simulink* simulation that is using this array.



Simulation Index: 533
Number of Simulations: 820
Fraction done of current Simulation: 0.4
Code: 4
SNR: 3.25
Symbols Done: 7.968e+008
Nuber of Errors: 8
Current BER: 1.004e-008
Collision: 1
Diff: 1
Skip to This Simulation: -1

Simulation parameters for the tester have to be defined in a 'simulation' variable in Matlab This variable has to be an array of strucures with the following fields

simulation =

1x16 struct array with fields.
nBits
nErrors
... (simulation paramters e.g., SNR, Spacing, Punture)

Each structure in this array specifies the parameters of one simulaiton The simulations (in the case above 16 simulations) will be executed sequentially A simulation is terminated (and the next simulation is started) if either 'nBits' symbols have been simulated or if 'nErrors' errors occured (in the later case, the simulation will run for another approximatelly 5 minutes).

* See 'set_simulation m' script for details

Results are saved into the following vectors:

symbols, errors, BER, enc_collision, dec_collision, enc_diff, dec_diff

Figure 5.5: Performance testbed front-end

5.4 Functional Sub-blocks

Many modules of the developed platform include sub-structures with the same functionality. To avoid re-implementing these structures in multiple contexts, we have designed a number of various functional sub-blocks. These sub-blocks were designed during the development of the communication library as needed and implement simple counters, latches, specialized RAM as well as a *FOR loop* controller.

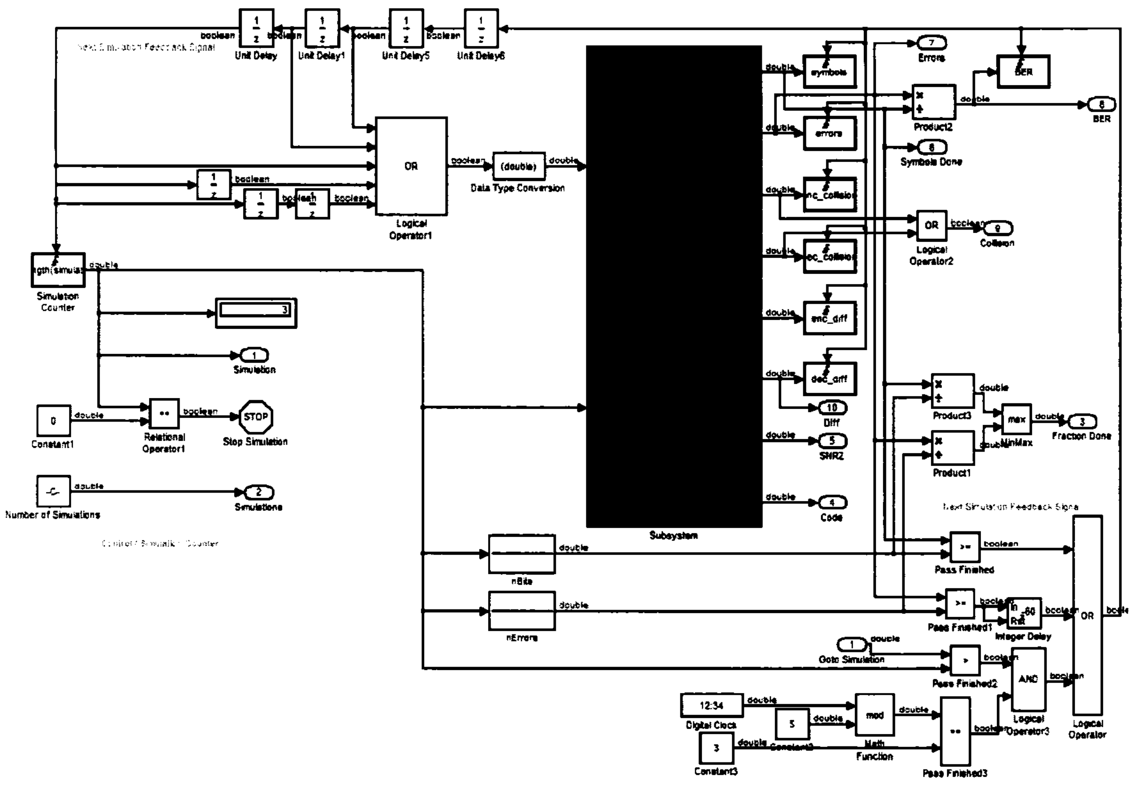


Figure 5.6: Performance testbed implementation I: Simulation control

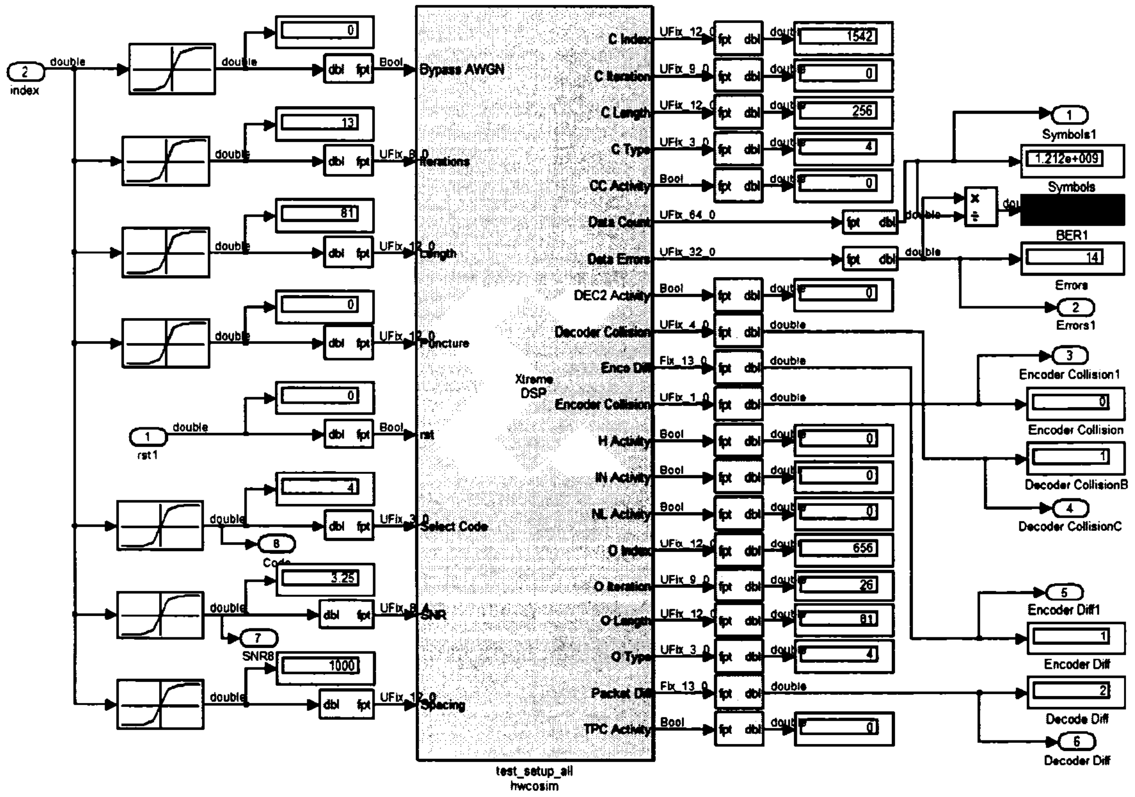
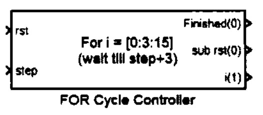
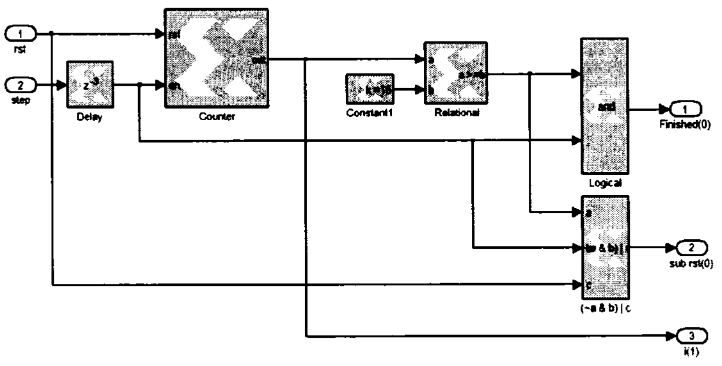


Figure 5.7: Performance testbed implementation II: Driving the FPGA



(a) *For Cycle Controller* block



(b) Implementation of the *For Cycle Controller* block

Figure 5.8: An example of a simple functional sub-block

The block in Figure 5.8 is an example of a simple utility block. Essentially, the block is a counter with some additional control signals. The block is extensively used in the implementation of the Viterbi decoder to loop over symbols, states and possible inputs. Please refer to Appendix B for further documentation of the utility blocks.

5.5 Chapter Summary

In this chapter, we have introduced in detail the components of our prototyping platform. We have described the modules contained in our communication library, the usage of these modules and the debugging, emulation and real time simulation tools included in the platform.

Chapter 6

Real-time Error Control Schemes

In this chapter, we will describe the following algorithmic modules that have been implemented using the developed modular library of communication system building blocks:

- Universal *linear block code encoder*,
- Universal *syndrome decoder,*
- Universal f*inite state machine encoder*,
- Universal *Viterbi decoder,*
- $(4,3)^4$ *Turbo product code* encoder,
- $(4,3)^4$ *Turbo product code* decoder.

The advantage of using our library is that we have developed modules that are highly customizable. The linear block code encoder and decoder can be set-up to use any linear block code and with a "push of a button," an FPGA programming file can be created from the description of the block code in *Matlab*. Likewise, the *finite state machine encoder* and the *Viterbi decoder* can be used with any convolutional code, which the user describes in a *Matlab* variable.

All encoder modules presented in this chapter expect a message packet on the input and output an encoded packet. The decoder modules work reversely. Some modules might impose limitations on input packet formatting (e.g., the *finite state machine encoder* module requires special formatting of the message packet; other modules work only on packets consisting of symbols of a specific alphabet).

Moreover, this chapter evaluates the performance of the above encoders in transmission over an AWGN channel. This was done to illustrate the usability of our platform and other proposed methods for simulating the multiuser environment can be applied in the future. Nonetheless, the AWGN model was not chosen arbitrarily as this model is appropriate for regular-power optical CDMA.

6.1 Linear Block Code Encoder and Decoder

A binary linear (n,k) block code is a code, whose codewords form a *k*-dimensional subspace of the *n*-dimensional vector space over the binary field *GF(2)* [46]. Therefore, any such codeword can be represented as a linear combination of the basis vectors $\mathbf{g}_1, \mathbf{g}_1, ..., \mathbf{g}_k$ of this subspace. The mapping (i.e., encoding) from a message **m** to a codeword **c** can be formulated as a matrix multiplication. The basis vectors of a *k*-dimensional vector over *GF(2)* can be chosen such, that the coordinates of a codeword in the abovementioned basis are equal to the message vector corresponding to that codeword. Thus, we can write

$$\mathbf{c} = m_1 \cdot \mathbf{g}_1 + m_2 \cdot \mathbf{g}_2 + ... + m_k \cdot \mathbf{g}_k,$$

where the codeword vector **c** is of length *n* bit, m_1, m_2, ..., m_k are the message bits and $\mathbf{g}_1, \mathbf{g}_1, ..., \mathbf{g}_k$ are the basis vectors defined above. Using matrix notation, this formula simplifies to:

$$\mathbf{c} = \mathbf{m} \cdot \mathbf{G},$$

where **G** is a *k-by-n* matrix formed from the basis vectors $\mathbf{g}_1, \mathbf{g}_1, ..., \mathbf{g}_k$. The matrix G is called the *generator matrix* of the block code.

Similarly, we will formulate decoding of a received vector **r**, which is the sum of the codeword vector **c** and the binary error vector **e**, using a matrix multiplication. Consider the basis vectors $\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_{n\text{-}k}$ of the null space of the subspace formed by code's codewords. Each of these vectors is orthogonal on the basis vectors $\mathbf{g}_1, \mathbf{g}_2, ..., \mathbf{g}_k$. If we multiply the received vector by one of the basis vectors $\mathbf{h}_i$, the terms including an inner product of $\mathbf{h}_i$ and $\mathbf{g}_j$ will be equal to zero:

$$\mathbf{r} \times \mathbf{h}_i^T = (\mathbf{c} + \mathbf{e}) \times \mathbf{h}_i^T = m_1 \cdot \mathbf{g}_1 \times \mathbf{h}_i^T + m_2 \cdot \mathbf{g}_1 \times \mathbf{h}_i^T + ... + m_k \cdot \mathbf{g}_k \times \mathbf{h}_i^T + \mathbf{e} \times \mathbf{h}_i^T = 0 + \mathbf{e} \times \mathbf{h}_i^T.$$

Rewriting the formula in vector form for *i* from 1 to *n-k*, we get:

$$\mathbf{r} \times \mathbf{H}^T = (\mathbf{m} + \mathbf{e}) \times \mathbf{H}^T = \mathbf{0} + \mathbf{e} \times \mathbf{H}^T = \mathbf{e} \times \mathbf{H}^T,$$

where **H** is a *n-k-by-n* matrix formed from the basis vectors $\mathbf{h}_1,\mathbf{h}_2,\ldots,\mathbf{h}_{n-k}$ and called the *parity check matrix*. We see, that $\mathbf{r}\times\mathbf{H}^T$, which is called the *syndrome*, is independent of the codeword vector and therefore, we can use it to determine the error vector by comparing it to pre-evaluated *syndromes* for all possible error vectors.

The formulation of encoding and decoding using matrix multiplications makes the algorithms suitable for implementation using our matrix multiplication module. In the following subsections, we describe the implementation of the encoder and the decoder using the *generator* and *parity check matrices* of the Hamming (7,4) code. Nonetheless the implementation is universal and the modules can be used to encode and decode any linear block code.

6.1.1 Linear Block Code Encoder

In the encoder implementation, we use the developed matrix multiplication module to multiply the input packet by the *generator matrix* of the block code in *GF(2)*. In case of the Hamming (7,4) code, the *generator matrix* is given by:

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

The encoding is done in two steps as seen in Figure 6.1. First the Multiply By Matrix module multiplies the message vector by the *generator matrix* over real numbers and then the Symbol Operation module takes the output modulo 2 to get the result over *GF(2)*. Note the use of the Slice block which implements the modulo 2 operation by retrieving the least-significant-bit of the symbol.

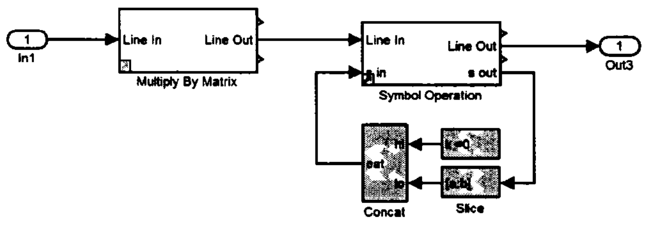


Figure 6.1: Schematics of the linear block code encoder

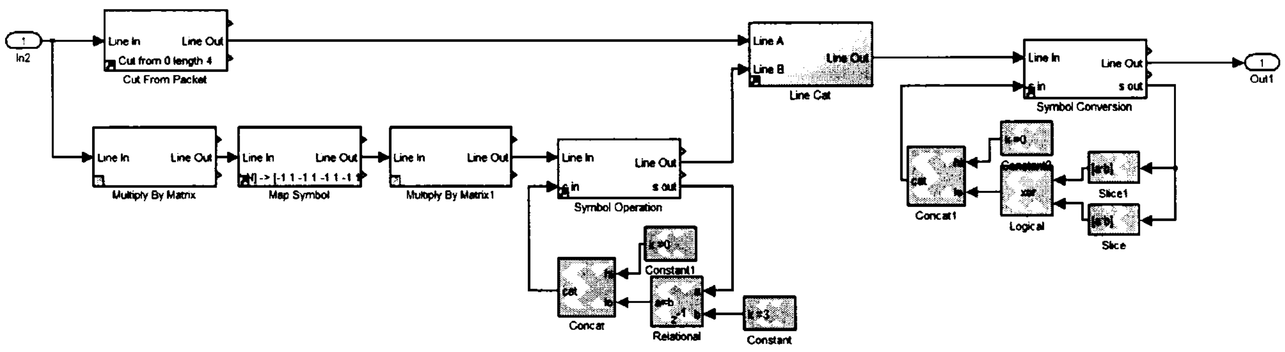


Figure 6.2: Schematics of the Hamming (7,4) code decoder

Our implementation of the Hamming (7,4) encoder uses 212 slices and 7 RAM blocks, that is 1.5% of the slices and 12.5% of the RAM available on an XC2V2000 device. Nonetheless, we are wasting most of the capacity of the RAM blocks. The 7 RAM blocks can be replaced with distributed RAM using less than 7 additional slices, as we are using the RAM blocks to store the columns of the *generator matrix* which are only 16 bits. Therefore, the module can be implemented using 219 slices, which is 2% of the resources of the XC2V2000 device. The module has a latency of 18 clock cycles and is able to process packets continuously. Generally, when implementing an (n,k) linear block code, the latency of the module will be $11+n$ and the amount of required resources will grow linearly with the size of the *generator matrix* of the code. The module will still be able to process packets continuously.

6.1.2 Linear Block Code Decoder

The implementation of the linear block code decoder uses the formulae derived earlier in this section. Specifically, the input vector (i.e., the received codeword corrupted by errors) is multiplied by the parity check matrix **H** and the *syndrome* is obtained. The *syndrome* is than mapped to the error vector which is added modulo 2 to the original message to obtain the corrected message.

In the implementation of the block decoder, depicted in Figure 6.2, we follow this procedure while using modules from our library. The decoding process consists of the following steps:

- First, the processing is split into two branches: in one branch, we evaluate the error vector, while in the other branch we store the systematic bits of the original message,
- Next, in the bottom branch, within the Multiply By Matrix module the message is multiplied by the parity check matrix **H** obtaining the error syndrome vector which is then mapped to *GF(2)* consisting of the elements +1 and -1 using the Map Symbol module,
- Consequently, the Multiply By Matrix1 module uses another matrix multiplication to convert the *syndrome* into an error vector,
- Finally, the error vector is added to the original received vector using the Line Cat and Symbol Conversion modules.

Specifically for the Hamming (7,4) code, the output of the first matrix multiplication by the parity check matrix

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

is a three bit vector which is mapped back to *GF(2)* analogously to the encoder. In this case, we form the field of the elements -1 and +1. This is done to enable the next operation, where vectors matched to the syndrome vectors are used to identify the positions of the erroneous message symbols. The mapping is done using the Map Symbol module which transforms the even results to -1 and the odd results to +1.

Now, if we take the inner product of the syndrome vector with another three bit vector consisting of -1 and +1, we will get the result 3 only if the vectors were equal. Therefore, if we multiply the syndrome of the received codeword by the syndrome representing an error in the first message bit, we will get three only if there is an error in the first bit. We are doing four such multiplications (for each of the message bits) using matrix multiplications by the matrix:

$$M = \begin{pmatrix} 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix}.$$

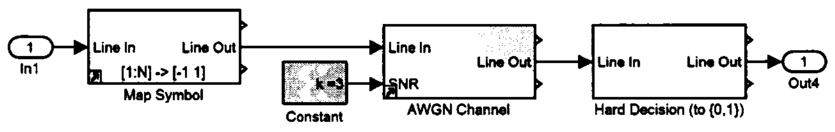


Figure 6.3: Modulation and channel estimation used for the Hamming(7,4) scheme

The resulting vector identifies the errors and the Line Cat and Symbol Conversion modules can be used to add the error vector to the original data.

The Hamming (7,4) decoder uses 467 slices and 9 RAM blocks. Seven RAM blocks are used by the matrix multiplication modules and can be replaced with less then 7 slices, as described in the encoder implementation, while 2 RAM blocks are used by the Line Cat module. Optimized to 474 slices and 2 RAM blocks, the decoder utilizes 4.4% of the slices and 3.6% of the Block RAM blocks available on an XC2V2000 device. The module has a latency of 46 clock cycles and can process a continuous stream of packets. The necessary resources scale with the length of the block code in a way similar to the encoder, i.e., a more powerful code, such as the Golay (23,12) code, would require approximately three times the resources of the Hamming (7,4) code.

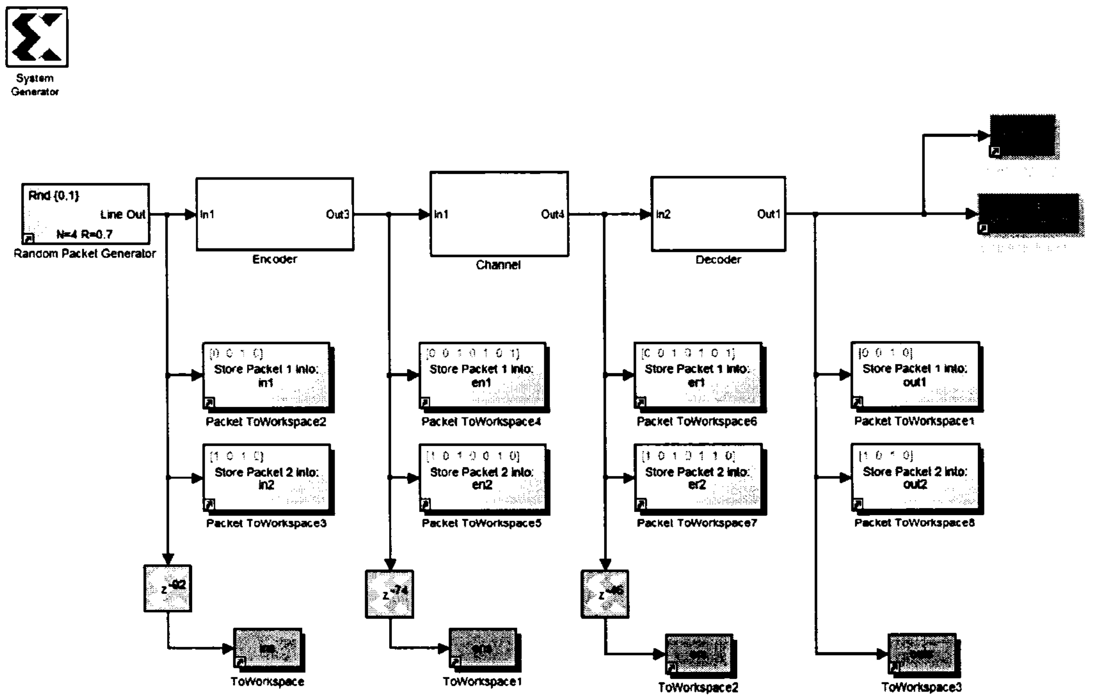


Figure 6.4: Testing of the Hamming (7,4) scheme in *System Generator*

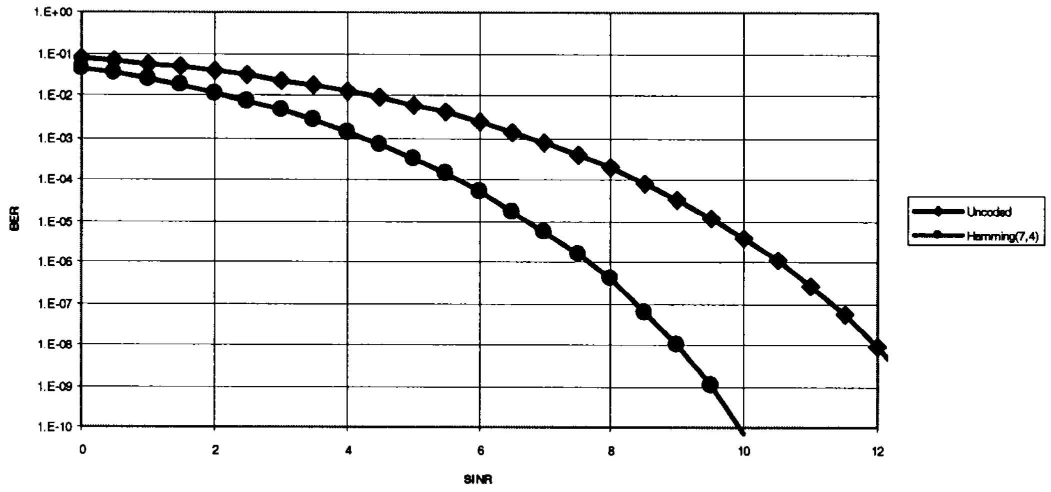


Figure 6.5: Bit error rate performance of the Hamming (7,4) block code

6.1.3 Communication System Setup and Testing

For testing and performance analysis, we have built a communication system using our developed encoder, decoder and a simulated AWGN channel implementation shown in Figure 6.3. We have tested our modules using the analyzer blocks provided for debugging purposes in our communication library. Figure 6.4 gives an example, where we have connected the communication system to several modules displaying the data while processed by the system.

6.1.4 Performance of Implemented Block Encoder and Decoder

This implementation of the Hamming (7,4) encoder and decoder is based on our parallel matrix multiplication implementation which is able to process a continuous stream of data. Therefore, with little additional processing, both the modules are able to process packets stacked one after another with space only for the parity check symbols and the packet header. The latency of the encoder is 18 clock cycles and the latency of the decoder is 46 clock cycles.

When implemented on a *Xilinx Virtex II* (XC2V2000) FPGA chip, the encoder and decoder can be clocked at 60MHz. If we evaluate the maximum transmission speed enabled by this processing, we have to consider that additional 3 packet header symbols will increase the size of a packet to 10. Therefore, the maximum transmission speed is $60 \cdot 7/10 = 42$ Mbps of encoded symbols and $60 \cdot 4/10 = 24$ Mbps of message data. The implementation of the encoder and decoder utilizes less than 7% of the resources of the above mentioned chip. Therefore, an increase in the throughput can be achieved by instantiating 14 encoder and decoder pairs within one chip. This will increate the throughput to 336 Mbps. Further increase in performance can be achieved by utilizing a larger or higher speed grade FPGA chip. An improvement of approximately 60% is possible with a -7 speed grade *Virtex II Pro* (XC2VP20) chip giving us a transmission rate of 538 Mbps, while a five-fold improvement can be achieved with the largest devices giving us gigabit speeds. This processing speed can be achieved with no custom optimizations by just directly implementing the textbook encoder/decoder algorithm. For systematic codes, the matrix multiplication within the encoder and decoder can be simplified by using the fact that a large portion of the *generator* as well as *parity check* matrices is an identity matrix. This would reduce the amount used resources allowing higher throughputs by increasing parallelization.

We have integrated the Hamming(7,4) code into our optical CDMA FPGA testbed, where the code was employed to counter errors introduced by multiuser interference. We have modeled the multiuser interference as having Gaussian distribution due to the *central limit theorem* [35]. We have run bit-error-rate simulations on the testbed at 24 Mbps and achieved a coding gain of several dB. The simulation results are depicted in Figure 6.5. The high speed of the simulation enables us to evaluate the performance of the decoder at over 20 different SINR levels with 208 billion simulated bits within approximately 3 hours.

6.2 Convolutional Codes Encoder and Decoder

We have used our communication library to implement a *finite state machine* encoder and a Viterbi decoder. We have constructed these modules while developing the communication library. The main purpose of these modules was to test our concepts and to enable implementation of a simple communication

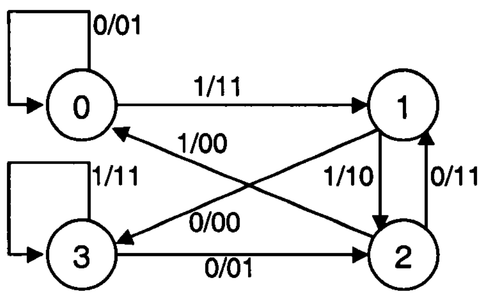


Figure 6.6: An example of a simple finite state machine

system. Therefore, the modules are not highly optimized. Particularly, the Viterbi decoder is a serial implementation of the textbook algorithm and therefore the maximum throughput of this module is quite limited. Nonetheless, the modules are fully customizable and demonstrate the flexibility of our platform. The encoder and decoder can utilize any finite memory code and the description of the used *finite state machine* is passed to the modules as a *Matlab* variable.

In the following subsections, we first give a brief review of convolutional codes using a *finite state machine* (FSM) and *trellis* diagrams as well as decoding using the Viterbi algorithm. We then describe the details of the implementation of the encoder and decoder.

6.2.1 Encoding and Decoding of an FSM Encoded Messages

A *finite state machine* is an automaton with s states that reads n input symbols (from an alphabet of size a) at a time and, depending on the current state and the input symbols, it updates the state and generates k output symbols. Such a *state machine* can be described by a *state transition* matrix **T** and *output* matrix **O**. The *state transition* is a s-*by*-a^n matrix which specifies the new state for each old state and combination of input symbols. The *output* matrix is a s-*by*-$a^n \cdot k$ matrix that consists of row vectors of k output symbols for each old state and combination of input symbols.

Depicted in Figure 6.6 is the representation of a simple 4 state *state machine* with a single binary input and two binary output symbols per transition. The Figure shows the *state machine* as a graph with states represented by nodes and transitions by edges. The "i/o" label of an *uv* edge specifies that the machine will go from state *u* to state *v* on the input vector **i** while outputting the vector **o**. We can describe the *state machine* using the matrices:

$$\mathbf{T}=\begin{pmatrix}0 & 1\\ 3 & 2\\ 1 & 0\\ 2 & 3\end{pmatrix} \text{ and } \mathbf{O}=\begin{pmatrix}0 & 1 & 1 & 1\\ 0 & 0 & 1 & 0\\ 1 & 1 & 0 & 0\\ 0 & 1 & 1 & 1\end{pmatrix}.$$

We can visualize the encoding of a codeword by starting in the "zero" state and following the edges according to the input. In this simple example, if we encode the message "1000", the *state machine* will go through states "0-1-3-2-1" and output the codeword "11000111". To enable efficient decoding, we want that the *state machine* terminates in state "zero". Therefore, we will append the output "1000" to the codeword corresponding to the transition to state 2 and finally state 0. We can visualize the encoding process by a path in a layered directed finite graph with symbols along the edges, as depicted in Figure 6.7. This graph is called a *trellis* and represents the state machine in time. Each node of the *trellis* can be identified by an (s,t) pair, where s specifies the state and t denotes the encoding stage.

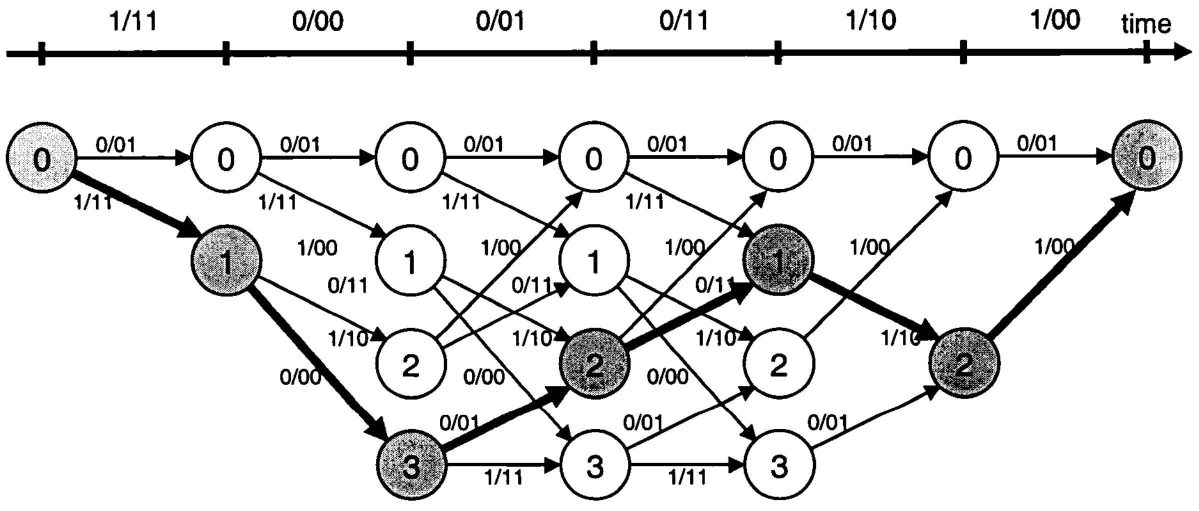


Figure 6.7: Trellis representation of FSM encoding

We will describe the Viterbi decoding algorithm [36] which can be used to decode a codeword encoded using a *finite state machine.* Having the channel observation information for a memoryless channel (e.g., AWGN, BSC, Rayleigh [26]), we first determine the probabilities for all edges of the *trellis* that the specific k-tuplet associated with an edge was sent in the specific stage. These probabilities give the transition probabilities for each edge. As each codeword corresponds to a path through the *trellis* from the node $(0,0)$ to the node $(0,l)$, where l is the total number of stages, the probability that a specific codeword was sent is equal to the multiplication of all probabilities associated with the edges along the path corresponding to that codeword. Replacing the probability p_e with $-\log p_e$ for each edge e, the multiplication becomes an addition. Therefore, if we assign a length of $-\log p_e$ to each edge, the total length of a path representing a codeword $\mathbf{c}$ will be equal to $-\log p_c$, where p_c is the probability that the codeword $\mathbf{c}$ was transmitted. We will maximize the probability p_c if we minimize the path length $-\log p_c$. Thus, determining the most probable codeword is equivalent to determining the shortest path through the trellis with the abovementioned lengths assigned to the edges.

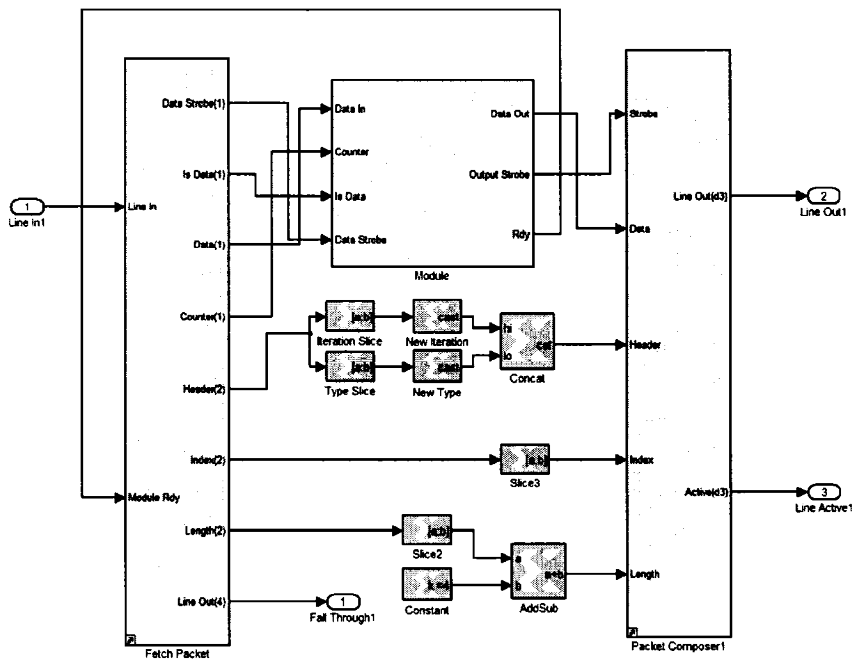
The shortest path through a *trellis* can be optimally determined in a time that is linearly growing with the number of nodes of the *trellis* using a simple *dynamic programming* approach. This approach is called the Viterbi algorithm.

Finite state machine **A** defines its parameters, state transitions and output on those transitions.

fsmA =	fsmA.Transitions =	fsmA.Outputs =
Transitions: [4x2 double]	0 1	0 1 0 1 1 1
Outputs: [4x6 double]	3 2	0 0 1 1 0 0
nStates: 4	1 0	1 1 1 0 0 0
nInSymbols: 2	2 3	0 1 0 1 1 1
nOutputs: 3		
nOutSymbols: 2		

Figure 6.8: *Matlab* specification of a finite state machine

(a)



(b)

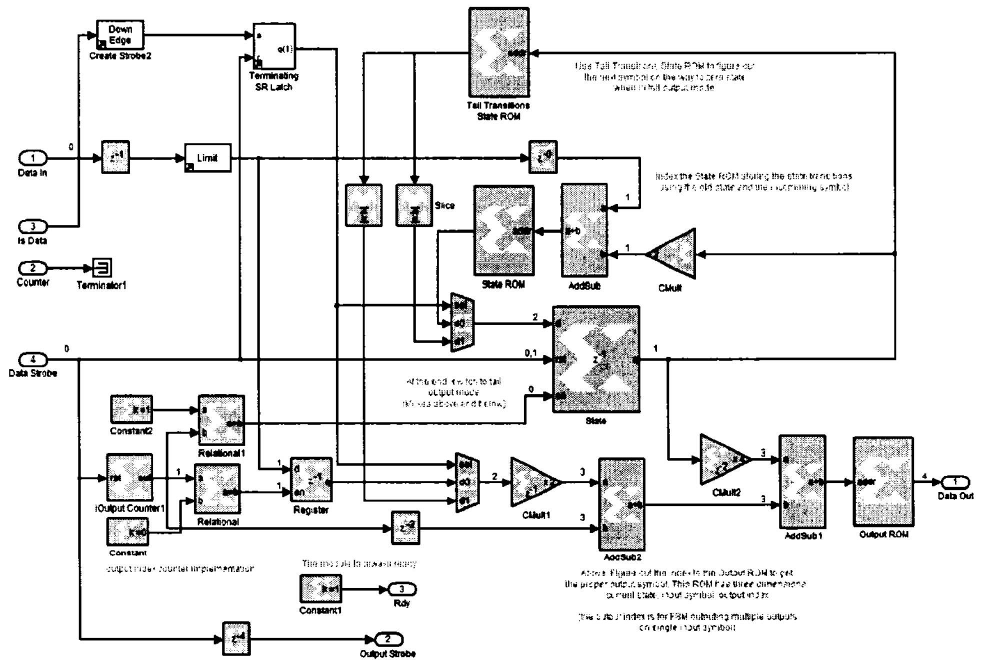


Figure 6.9: Implementation of the FSM encoder

(a) High level module schematic
(b) Functional schematic contained within the block in (a) named Module

6.2.2 Finite State Machine Encoder Implementation

While the implementation of the finite state machine depicted in Figure 6.9 is quite straightforward, we are putting stress on customizability. The description of a specific finite state machine is presented to the module as a *Matlab* parameter, which defines the state transitions and output vectors. An example of the definition of a small state machine can be seen in Figure 6.8. This universal module then emulates the supplied state machine.

The size of the input alphabet is automatically determined using the supplied state machine description by counting the number of transitions defined for a specific state. If the size of the input alphabet is **N**, the input symbols have to be binary representations of numbers from zero through **N-1**. There are no limitations on the output alphabet and on each transition, the encoder can output any symbol limited only by the number of bits provided for the output.

A sequence of terminating symbols is appended to the codeword. This sequence ensures that the state machine terminates in the zero state. We have implemented a *Matlab* function that automatically computes a set of optimal terminating sequences for the given state machine. This function is automatically called in the initialization of the module. We impose two conditions on the state machine:

- If the state machine is in state *zero* and a *zero* symbol is received, the machine must stay in state *zero,*
- At least two output symbols have to be generated for each input symbol.

The terminating sequences for different states can consist of a different number of symbols. The first condition allows us to properly terminate all codewords by padding all sequences with *zeroes* to the length of the longest terminating sequence. The second condition is due to implementation. The heart of the implementation is a loop that consists of a register storing the current state and a memory for lookup of the next state. We cannot remove either of these elements from the implementation. This loop has a latency of two and therefore the state can be updated only once in two clock-cycles. Therefore we impose the condition that at least two outputs have to be generated for each input. This gives us enough time to update the state. To simplify the implementation of the encoder, the module requires that the input symbols are spaced by dummy symbols according to the number of output symbols per input. For details and the implementation of the spacing using a library module, see Appendix B.

The implementation in Figure 6.9(a) instantiates the design necessary for the module to work within our developed platform environment. Figure 6.9(b) depicts the design contained within the *Module* block. The core of the encoder is the state register that stores the current state of the state machine. With each new input symbol, this register is updated using a look up into the transition matrix which is stored in a ROM. At the same time, an output is generated using a look up table. After receiving all input data, the module switches to tail output mode and appends a tail sequence to the codeword. This tail sequence terminates the state machine in the zero state. The tail sequence tree as well as the optimal tail length is determined automatically in the module initialization using a *Matlab* function. This function implements a dynamic programming algorithm that finds the shortest sequence leading to a zero state for each state. These sequences do not necessarily have the same length, therefore all sequences are padded with zeroes to the length of the longest tail sequence. The padded sequence has to terminate in zero state, this can be achieved easily if we place a limitation on the finite state machine saying that the machine has to stay in zero state if it was in zero state and the received symbol was zero.

With a simple 4 state *finite state machine* the encoder uses 136 slices and 3 RAM blocks. The requirements on the logic will not increase with the size of the *state machine* or the length of the codeword, whereas the memory requirements will grow with the size of the transition matrix. The encoder is able to process a continuous stream of packets for a rate $1/2$ (or less) convolutional code, as discussed above.

6.2.3 Viterbi Decoder

The Viterbi Decoder is implemented with the same universality as the finite state machine encoder and it uses the same *Matlab* description of the state machine as the definition of the used code. We implement the decoder using the textbook dynamic programming Viterbi algorithm [36]. Depending on the considered channel, each edge in the trellis representation of the code has to be assigned an additive measure, which is proportional to the probability of the transition represented by the edge. This assignment is done by a block within the implementation that has to be customized for different channel model (e.g., binary erasure channel, AWGN with a given estimated SNR).

The implementation is fully sequential with 4 nested loops: a loop over all symbols of the codeword, a loop over all states, all possible symbol values and over the multiple transition outputs. Trellis stage by trellis stage, the Viterbi algorithm computes the smallest cost path through to trellis. The core of this process with the RAM block storing the minimal distances achieved for each state is depicted in Figure 6.10. In the process, additional back-track information is stored and used afterwards to recreate the smallest cost path and output the decoded codeword. This backward pass of the algorithm is depicted in Figure 6.11.

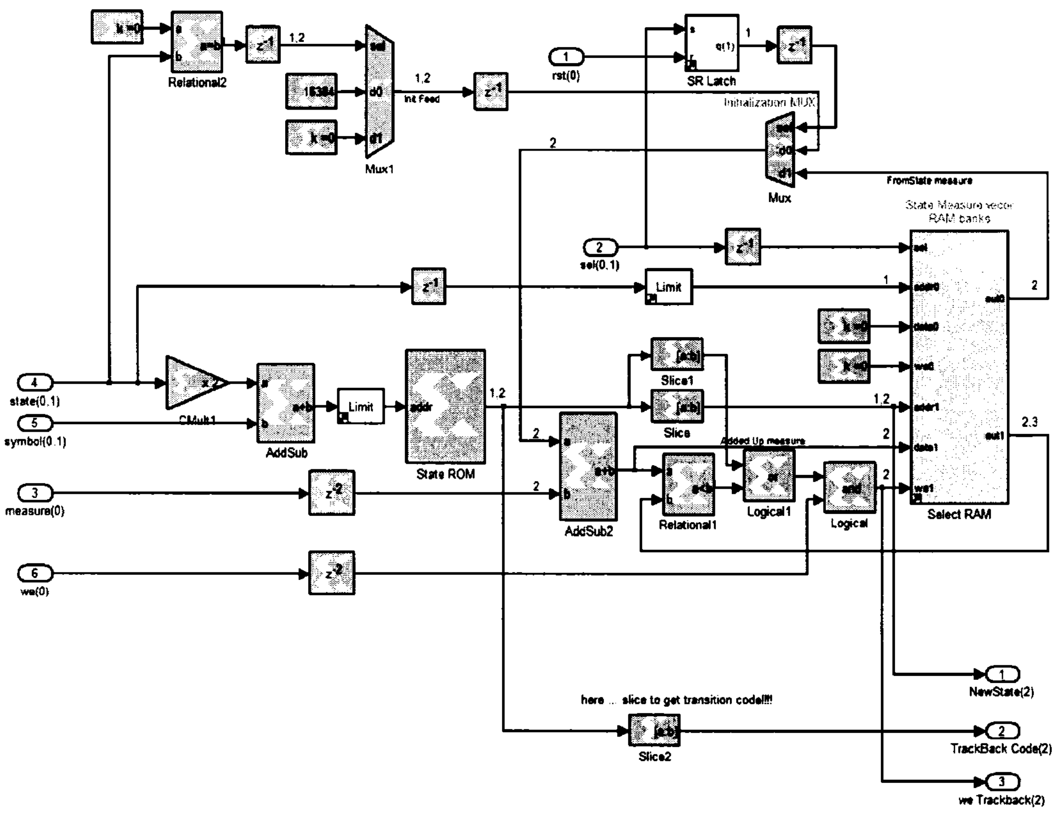


Figure 6.10: Core of the forward pass of the Viterbi algorithm

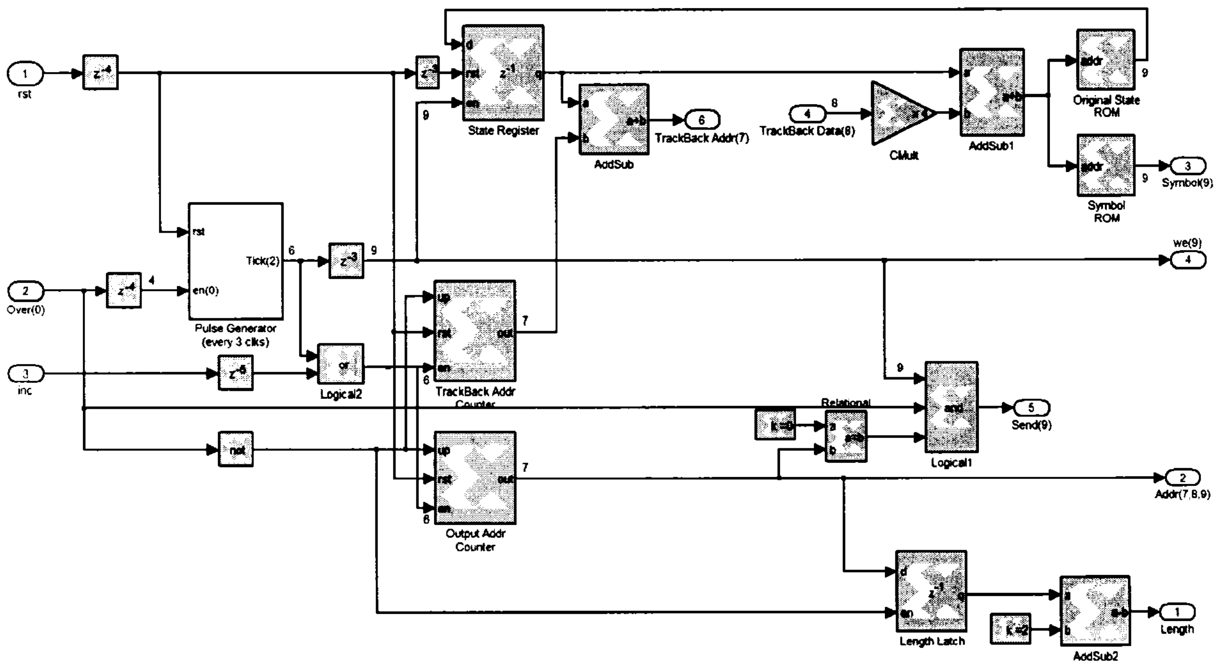


Figure 6.11: Backwards (track-back) pass of the Viterbi algorithm

The track-back information is stored compressed to the minimal number of bits needed to store which state preceded a specified state. This number depends on the maximum number of transitions that merge into one state and is automatically evaluated within a *Matlab* function [tc,ts,ty] = StateTransitionCodes(fsm). A look-up table, automatically generated by the mentioned *Matlab* function, is utilized to get the previous state from the current state and the transition code during the track-back process.

For details on the implementation of this decoder, please refer to Appendix B. For a simple 4 state *finite state machine*, the decoder uses 406 slices and 9 RAM blocks. The logic requirements will stay constant for any size of the state machine and any codeword length, whereas the memory requirements will grow quadratically with the size of the state machine and linearly with the length of the codeword. The processing speed will decrease linearly with the size of the state machine.

6.2.4 Communication System Setup and Testing

Using a few simple packets and a simple state machine, we have verified general functionality of our modules. This has been done using a packet generator with the encoder, decoder and an AWGN channel module in a setup very similar to the scheme used in Figure 6.4. More extensive testing has been done during performance simulations described in the next paragraph.

6.2.5 Performance of our Implementation of Convolutional Coding

The *Finite State Machine* encoder module is able to process a continuous stream of packets, while the *Viterbi Decoder* module requires considerable time to process a packet. Therefore, the bottleneck of a system implemented using these modules would be the *Viterbi Decoder.* We have tested the *Viterbi Decoder* using a trivial 4 state *finite state machine* code using a message length of 81 bits, where the decoder was able to process all the data only if the spacing of the packets was above approximately 3000 symbols. At 50 MHz, this gives us a maximum throughput of approximately 1.35 Mbps. As processing speed decreases linearly with the number of states of the used *state machine*, a *finite state machine* with 256 states would achieve a throughput of only about 40 kbps.

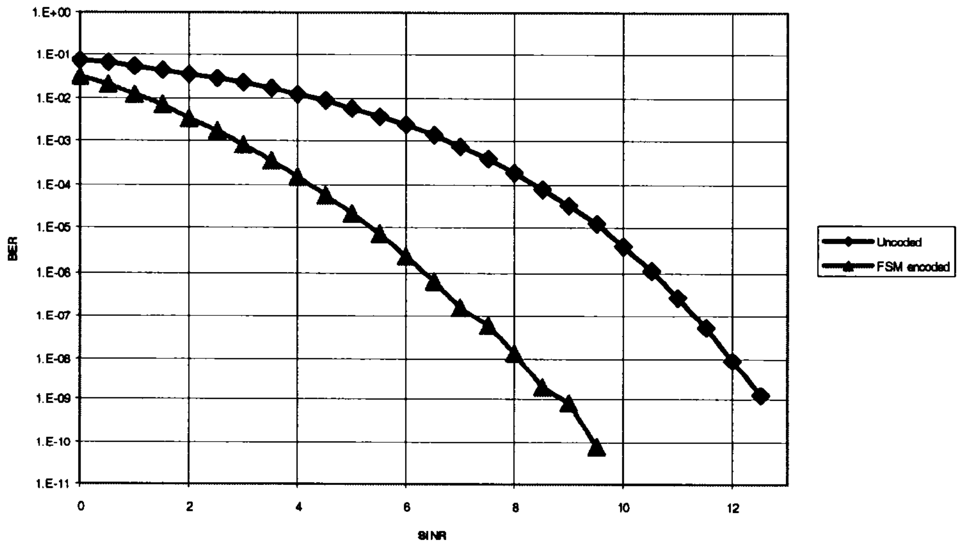


Figure 6.12: Bit error rate performance of a simple 2 state FSM code

We have used our automated testbed to run the performance simulations of the system employing the *Finite State Machine* encoder and the *Viterbi Decoder.* The throughput estimates are based on counting the packets lost due to exceeding the maximum throughput of the *Viterbi Decoder.* Figure 6.12 shows the simulated performance of the following FSM code (the code from Figure 6.6 is catastrophic, as we discovered after doing simulations):

$$\mathbf{T} = \begin{pmatrix} 0 & 2 \\ 0 & 2 \\ 1 & 3 \\ 1 & 3 \end{pmatrix} \text{ and } \mathbf{O} = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}.$$

6.3 $(4,3)^4$ Turbo Product Code

A *product code* is a code that utilizes simple linear block codes to encode a substantially longer codeword. The simplest type of a *product code* can be visualized as a two dimensional grid of symbols which is encoded row-wise and column-wise using a linear block code. We can decode such a codeword using iterative processing; first decoding the rows – correcting some symbols; then decoding the columns and again correcting some additional symbols; finally repeating this process. We call codes that can be decoded in this way *turbo*

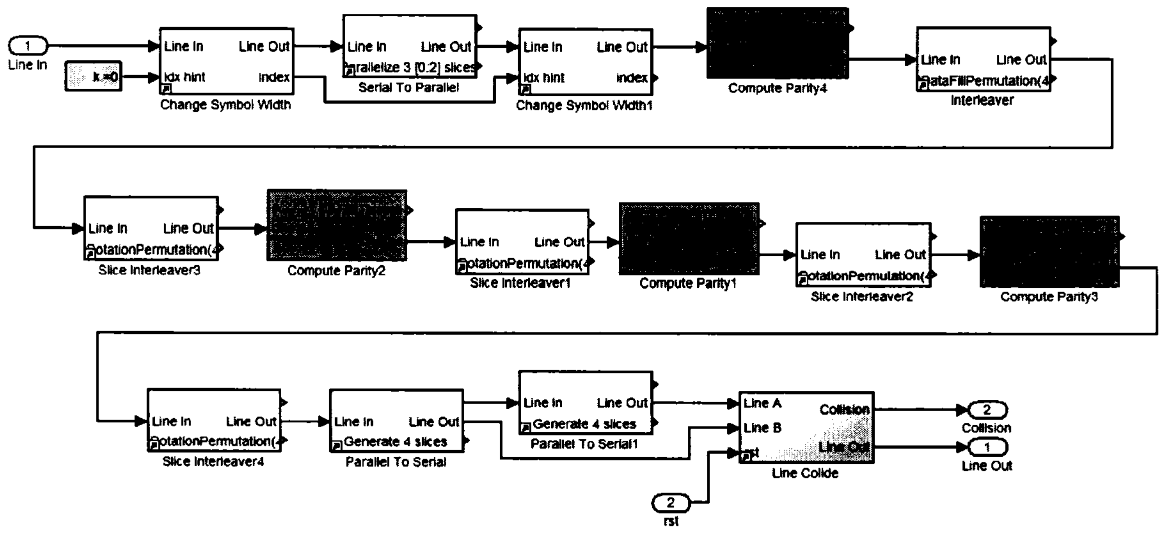


Figure 6.13: Turbo Product Code encoder implementation

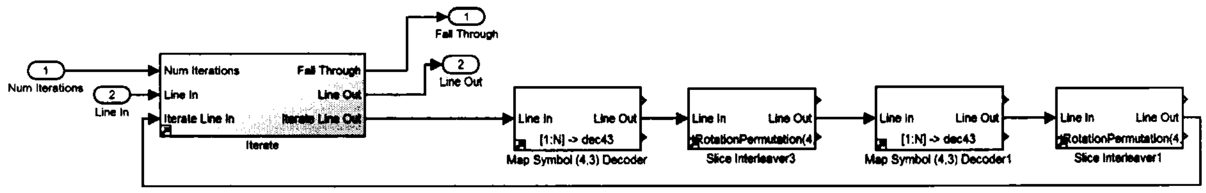


Figure 6.14: Iteration loop

product codes. Good performance and lower computational cost (compared with *turbo convolutional codes*) makes *turbo product codes* attractive for solutions, where high speed or low power consumption is required [39].

In this section, we present the implementation of a four dimensional (4,3) parity check *turbo product code.* The codeword length of this code is 256 with 81 information bits and 175 parity check bits and the code can be visualized as a four-dimensional hypercube.

6.3.1 Encoder of the Turbo Product Code

The encoder implementation is straightforward and simple. We split the 81 bit message into triplets which we extent by one bit into quadruplets as we pad the bit vector with zeroes to a size of 256 bits. Consequently, we compute the parity bits in each of the four dimensions. We do this by computing the parity in one dimension, applying a permutation that shifts dimensions (i.e., rotation of the four dimensional hypercube) and repeating this process four times. We use the same dimension-shifting permutation in the decoder.

The encoder uses 2107 slices and 13 Block RAM blocks. This corresponds to roughly 20% of the resources available on a XC2V2000 device and 15% of the resources on an XC2V3000 device.

6.3.2 Decoder of the Turbo Product Code

The turbo decoder operates by utilizing a soft-input soft-output decoder along each of the four dimensions. The decoder, depicted in Figure 6.14, executes numerous iterations running the decoding along each dimension multiple times. With each iteration, the side information provided by decoding other dimensions improves the results of the decoder along a certain dimension.

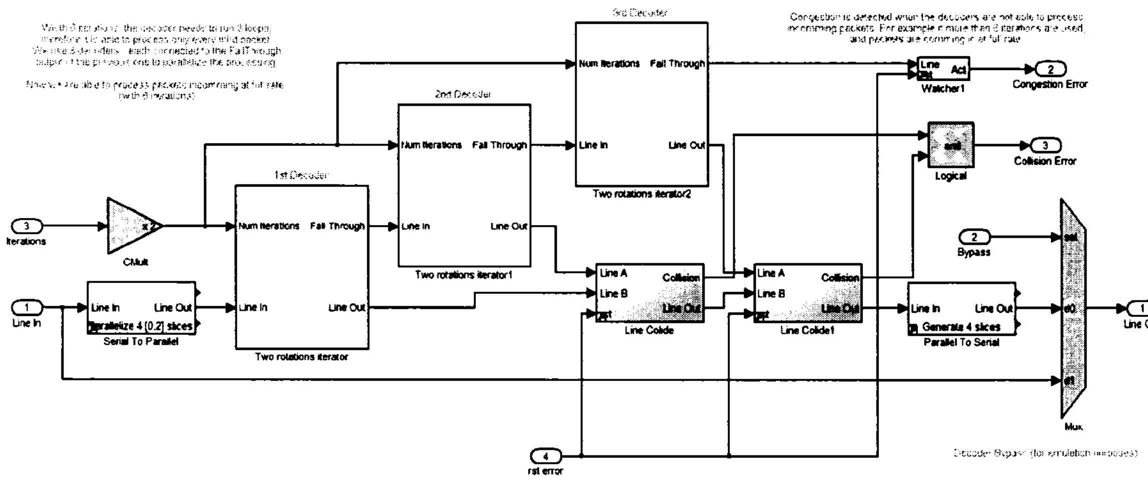


Figure 6.15: The *Turbo Product Code* decoder

The core of the algorithm is the soft-input soft-output parity check code decoder implemented within the *Map Symbol (4,3) Decoder* block in Figure 6.14. The overall performance and speed of the decoder is directly proportional to the performance of this component decoder. Therefore, high optimization of this decoder is desired. In our implementation, we are using a look-up table to implement the decoder, as this turned out to be the fastest implementation possible. However, such an implementation is only possible for small codes, such as the four symbol parity check code utilized here. We use a table computed using the *BCJR* algorithm during design time, which stores the decoder output for all possible inputs for a fixed SNR. With reasonable quantization of the input values, the whole table can be stored in memory. Thus, the decoding process takes only one clock cycle per constituent decoder

The parity check decoder processes the 256 bit vector in quadruplets. After the decoding along specific dimension, a dimension-shifting permutation (as explained in the encoder subsection) implemented in the *Slice Interleaver* blocks in Figure 6.14 is applied to the vector to rearrange the bits, so that bits along the next dimension are placed next to each other in the vector. This enables easy decoding along the next dimension. The time the decoder needs to decode one packet is dependent on the number of decoding iterations.

The implementation of the decoder is depicted in Figure 6.15. This design parallelizes the *Two rotations iterator* (which decodes along two dimensions) depicted in Figure 6.14. This setup increases the number of iterations we will be able to execute at the required processing speed. If the decoder is not able to process the packets at the supplied speed, it will signal a congestion using a special error output.

The decoder uses 3024 slices and 37 RAM blocks, which is about 28% of the slices and 66% of the RAM blocks of a XC2V2000 device and 21% of the slices and 39% of the RAM blocks of a XC23000 device.

6.3.3 Communication System Setup and Testing

We have embedded our encoder and decoder in a comprehensive testing setup, depicted in Figure 6.16. This setup enables us to test the communication system with different channels, various puncturing scenarios and customizable number of iterations. We have connected the system to a simulated AWGN channel. The setup also enables us to choose the transmission speed and monitor the congestion error output of the decoder.

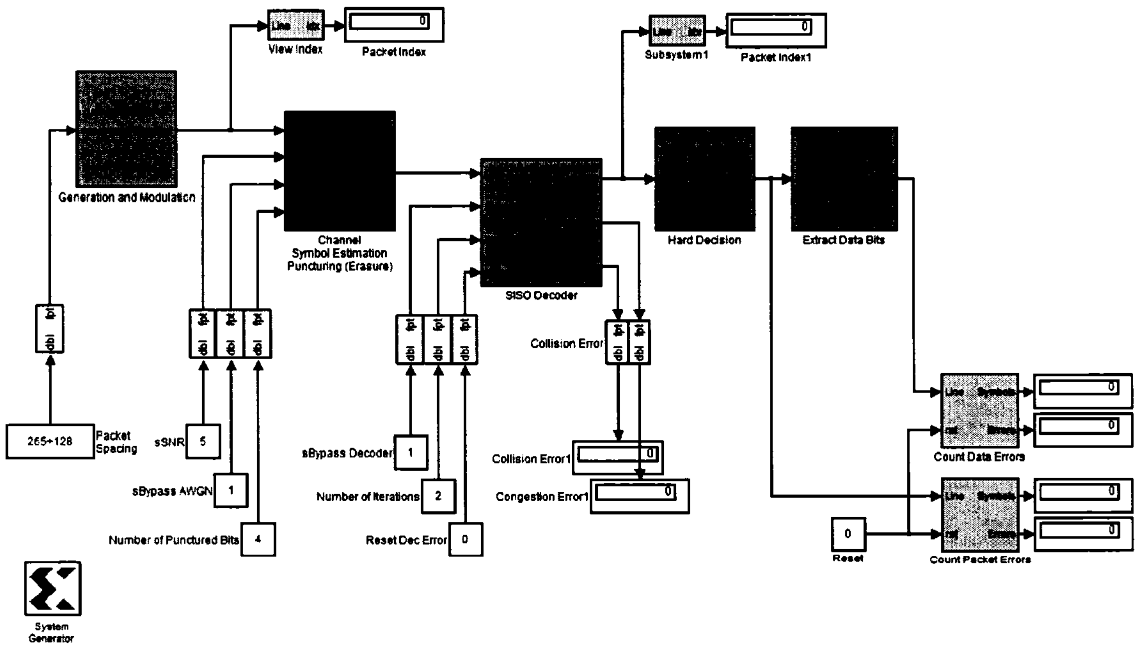


Figure 6.16: Testing setup of the TPC decoder

6.3.4 Performance of our Implementation of a Turbo Coded System

Utilizing our automated testbed, we have determined the performance of the turbo coded system for various channel conditions, puncturing scenarios and number of iterations used in the decoding process. The testbed enabled us to simulate 902 billion bits in more than 800 different setups in one overnight session. Figure 6.17 depicts the performance of our code for various channel *signal-to-noise* ratios while utilizing 3 iterations in the decoder. As seen from, Figure 6.17 which depicts the performance of the code depending on the number of iterations, this number of iterations is sufficient, as there is only little improvement in the performance of the code after the third iteration. Note that, one iteration includes decoding along all four dimensions of the code and therefore three iterations will include 12 constituent decoder passes.

Furthermore, we have analyzed the influence of puncturing on the performance of our code. From Figure 6.19 we can see that the performance of our code decreases rapidly with the number of punctured bits. Nonetheless, on channels with high SNR, this technique can be used to increase the throughput of our coding scheme.

Utilizing our decoder, we have been able to process data continuously with 3 decoder iterations at 66 MHz giving us a speed of 66 Mbps of coded transmission. With some optimization of the RAM usage, or employing external RAM, we can instantiate three decoders on a single XC2V2000 chip giving us a speed of 198 Mbps. Further increase in performance can be achieved by utilizing a higher speed grade or a biger FPGA chip. An improvement of approximately 60% is possible with a -7 speed grade FPGA giving us a transmission rate of 316 Mbps, while a five-fold improvement can be achieved with the largest devices giving us speeds just below 1 Gbps. To determine the message transmission speed we have to multiply the speeds given above by 80/256.

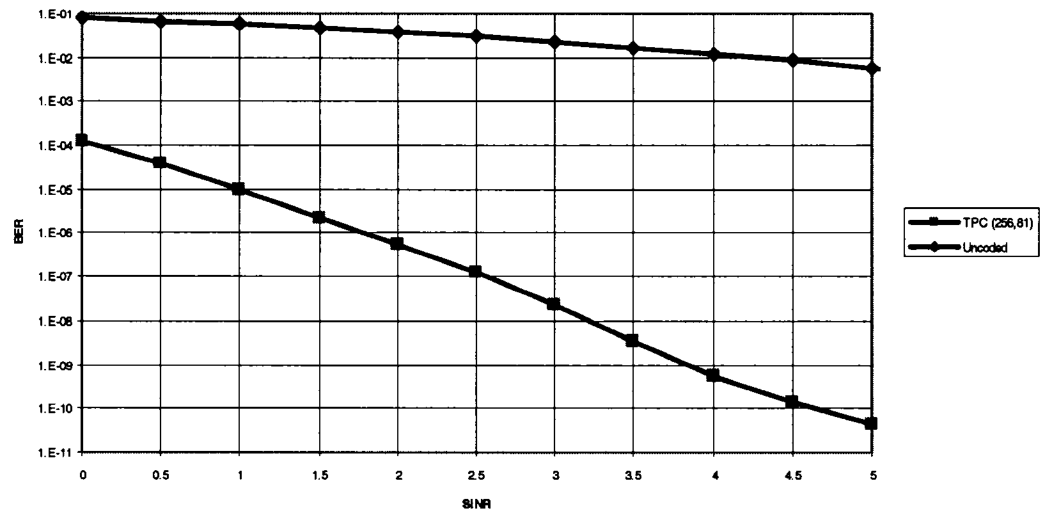


Figure 6.17: BER performance of our Turbo Product Code implementation

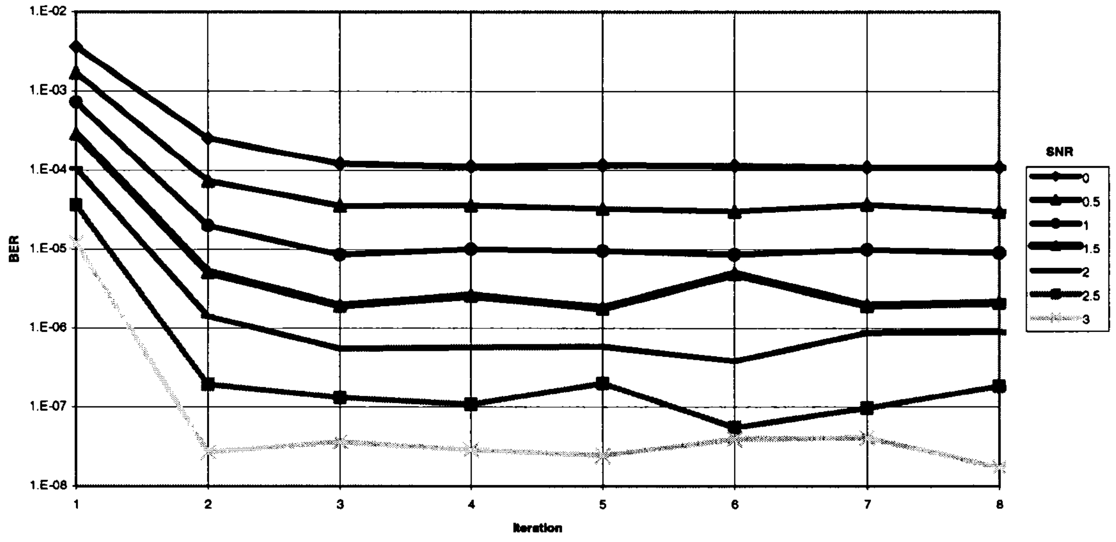


Figure 6.18: Convergence of the Turbo Product Code

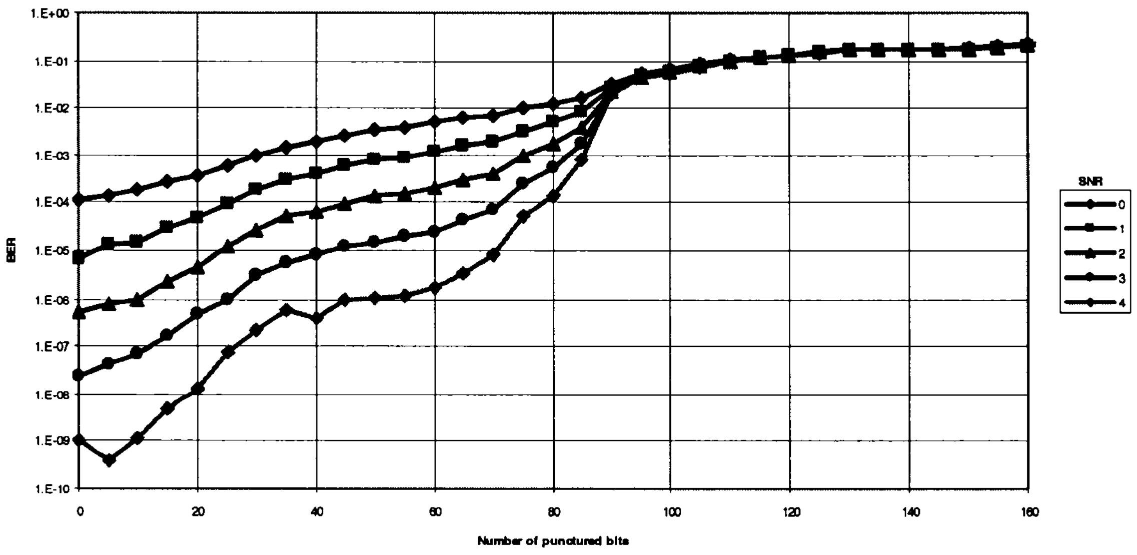


Figure 6.19: Performance of the Turbo Product Code with puncturing

Table 6.1: Comparing the performance of commercially available and our decoders

Algorithm	Speed	SLICES	BRAM	Max Speeds[2]
Our Hamming (7,4) Decoder	24 Mbps	474	2	0.5-2 Gbps
Our Viterbi Decoder	≈1 Mbps	406	9	50 Mbps
Our Turbo Product Code	20 Mbps	3024	37	0.5 Gbps
Reed Solomon1 (Xilinx)	722 Mbps	764	2	20 Gbps
Viterbi Decoder1 (Xilinx)	10-150 Mbps	805-3310	2-172	3 Gbps
Turbo Product Code1 (Xilinx)	123 Mbps	3313	17	3 Gbps
Turbo Convolutional Code[1] (Xilinx)	6.2 Mbps	1618	47	100 Mbps

[1] For details on these codes, please consult Table 2.3

[2] Theoretically achievable approximate speeds by parallelizing the design on large FPGA devices

6.4 Performance of Implemented Codes in OCDMA Network

Previously in this chapter, we have evaluated the performance of our implemented error-control schemes in terms of BER for a given signal-to-interference-and-noise ratio (SINR). The SINR is the sum of the channel noise and the multiuser interference. In the case of optical CDMA transmission, the effects of noise are usually negligible compared to the effects of multi-user interference. Hence, using the cross-correlation characteristics of a specific OCDMA spreading code, the SINR performance can be used directly to determine the number of simultaneous network users operating at a given BER.

The performance of our three implemented coding schemes has been determined for a specific two dimensional optical CDMA spreading code that was designed in [44] and [45]. In particular, we have used a 41-by-32 balanced code with differential detection and maximum normalized cross-correlation of 0.0991. The resulting performance is shown in Figure 6.20. We observe, that if the required target BER is 10^{-9}, our coding modules will allow us to increase the number of users from 40 to 70 for the Hamming code, to about 90 for the convolutional code and more than 250 for the (256,81) turbo product code.

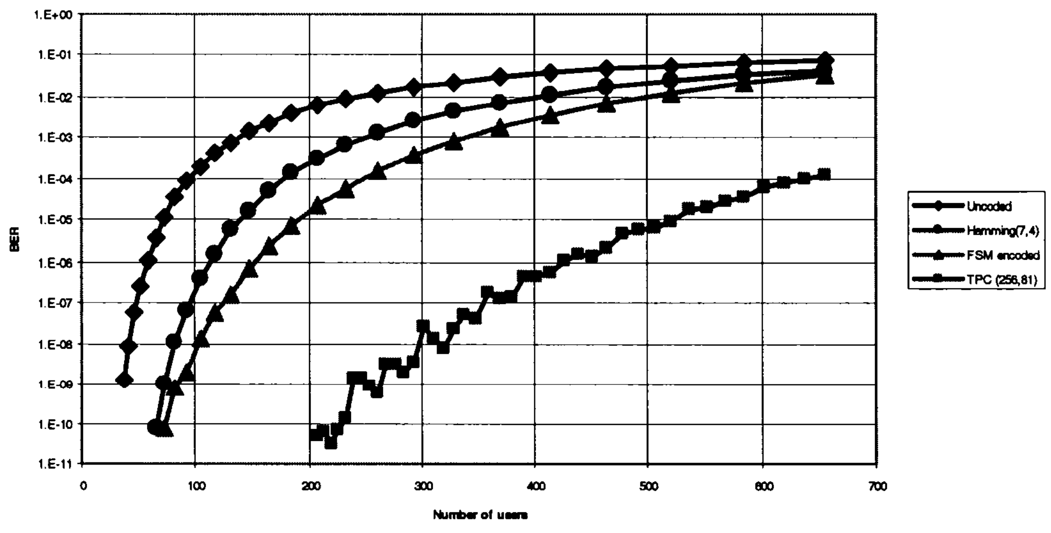


Figure 6.20: Improvement in the number of active users for OCMA network transmission based on different error-control codes and the BCDD (41,32, 0.0991) optical spreading code[44], [45]

6.5 Chapter Summary

In this chapter, we have introduced the implementation of three different error-control schemes: Block codes, convolutional codes and turbo product codes. We have described the implementation details for these schemes, explained testing mechanisms that were used to test them and evaluated their performance via simulations. In all cases, we have simulated transmission of approximately 10 Tb (10^{13} bits). Table 6.1 compares the throughput performance of our decoders with the commercially available decoders. We see that the commercially available solutions should perform better. Nonetheless, our modules were implemented with only basic optimization and demonstrate both usability and flexibility of our platform for further development of error control codes for optical CDMA systems.

Chapter 7

Conclusion

7.1 Summary of Research Achievements

In this work, we have presented an implementation of a prototyping platform for development of error control codes to be applied in an optical CDMA network prototype. In Chapter 2 we provided a review of hardware development techniques with a focus on *System Generator* from *Xilinx* and the description of the actual hardware/software setup of the platform. The presented prototyping platform was developed entirely with no prior setup in FPGA development. At the beginning of this thesis, we had to choose the hardware as well as the development environment. These decisions were not always straightforward; we have considered various hardware as well as tested several software packages that would eventually turn out not to be useful for our project.

Chapter 3 proposed a system architecture for the design of customizable modules for error-control coding. Chapter 4 presented the implementation of a new utility module that implemented automatic schematics generation. In Chapter 5, we provided a full description of our platform including debugging modules and testing concepts. Finally, Chapter 6 demonstrated the functionality of our platform by developing, verifying and testing three different error control schemes.

Hardware design is a huge field where the researcher often focuses on a narrow set of tools. Therefore, we have tried to provide an overview of all concepts being used as well as offer an in depth description of our research.

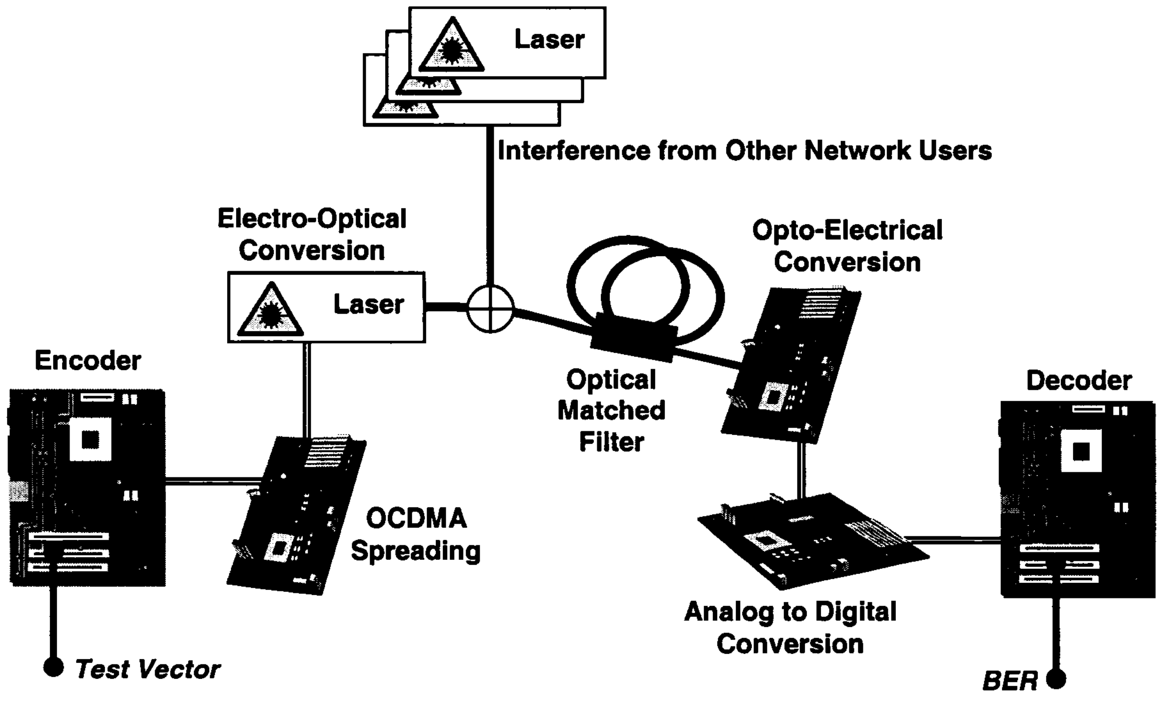

Laser
Interference from Other Network Users
Electro-Optical Conversion
Laser
Opto-Electrical Conversion
Encoder
Optical Matched Filter
Decoder
OCDMA Spreading
Analog to Digital Conversion
Test Vector
BER

Figure 7.1: Setup of an optical CDMA prototype

7.2 Future Work

The development of this platform is a step towards the implementation of a standalone optical CDMA network prototype. In this phase of the project, the whole communication system including a channel model was implemented on a single FPGA. In the future, through several setup stages, the channel model will be implemented in more detail, hardware will be separated into the encoder and decoder part and simulated external elements (e.g., fibre, electro-optical conversion) will be replaced with real photonic hardware.

Figure 7.1 depicts the schematic of the optical CDMA prototype with the current platform implementing only the encoder and decoder. In the future, the following elements will have to be added to the system setup:

- High speed interconnection of the FPGA hardware and the fibre-interface,
- Accurate and fast electro-optical as well as opto-electrical conversion,
- Fibre connection and optical matched filtering for decoding of OCDMA data.

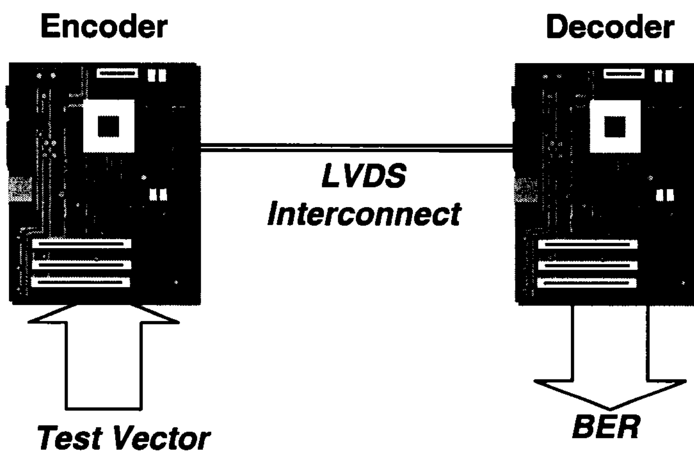


Figure 7.2: Back-to-back encoder and decoder setup

Most of these components can be tested independently by connecting the encoder and decoder implementations back-to-back, omitting “in-between” elements (e.g., optical fibre) or by connecting the decoder to external random packet generators and error counters. Future work includes separating the encoder and decoder as depicted in Figure 7.2. This step is important to prove that the hardware being used enables external communication at the required data rates.

Hardware requirements for the high speed interconnection and processing have been already assessed. We suggest using *low voltage differential signalling* (LVDS) in the interconnection of the FPGA hardware and the conversion elements. The use of high performance boards from AVNET that support this interface which will provide the FPGA with sufficient communication bandwidth is considered.

In future work, we plan to address the performance of our error control scheme. The developed Turbo Product Code scheme can be improved by application of larger parity check codes and by irregular interleaving. The use of other error-control methods [36] is also being considered. Nonetheless, in [19] we showed the inherent intractability of a class of path-counting algorithms, which are therefore not suitable for error-control coding implementation without further simplifications.

Appendix A

Workstation Setup

A.1 Introduction

This appendix describes the setup of the workstation used to develop the prototyping platform described in this thesis (the same setup is also necessary for usage of the platform) and should serve as a guide to the whole process of setting-up such workstation. Therefore, we will not only provide step-by-step installation instructions but also information on how to obtain the necessary software and hardware and where to look for additional information. Specifically, we will describe:

- software and hardware acquisition,
- software/hardware requirements and setup,
- intellectual property and other resources necessary to effectively use the platform.

We concentrate only on resources using *Xilinx* based FPGA solutions as opposed to Altera or other manufacturer solutions. Most hints on obtaining the software and hardware are specific to university environment.

A.2 Preparing for Hardware Development with *Xilinx*

First, you will require a development board with an *FPGA* chip. Such a board can be ordered through the *Xilinx* online store and costs several thousand dollars. We have used the *Nallatech Xtreme DSP Kit* which costs about CAN $3000.

Order through *Xilinx* as they are selling this kit below the *Nallatech* list price. Note that, other boards are also compatible with our development platform.

Next, you need to create an account on the *Xilinx* website. Go to http://www.xilinx.com/univ/ for the *Xilinx University Program* and click *Sign-up for access.* Creating this account is essential for obtaining access to parts of the *Xilinx* hardware development software suite and *intellectual property* cores. Having obtained an account, go to the professors-and-staff section of the *Xilinx University Program* and ask for the donation of the software packages and intellectual property listed below in the setup instructions of our workstation. You can obtain all this software free-of-charge.

A.3 Workstation Installation

In this section, we describe installation of our workstation using the *Nallatech Xtreme DSP Kit* development board. The system can be installed only on a PC running Microsoft Windows 2000 (or higher) with at least 1GB of available hard-drive space (2GB or hard-drive space and 512MB of RAM are recommended). Software must be installed in the order given below. The given versions of the software packages must be used. Different versions of the mentioned software might be required with other versions of *System Generator.* If you do not plan to use *System Generator* version *6.2i*, please refer to the *System Generator* online reference [43] for compatible software. To install the workstation:

- Install Matlab R13 (earlier versions will not be sufficient!).
- Install ISE 6.2i (both CDs). This is a *Xilinx* design environment and *Xilinx* is willing to donate the software for university purposes.
- Prepare the board. During the installation of the following items, the board will need to be connected to the USB port of the host PC. Windows will recognize it as a USB device and you will be asked to supply a driver from the FUSE CD.
- Install FUSE. This is a *Nallatech* board programming software, which is included with the *Xtreme DSP Kit.* During installation, you will be asked to connect the board to the USB port.

- Install *Xtreme DSP Development Kit.* This will plug into FUSE and install drivers for the specific hardware. The installation CD is provided with the kit.
- Install *System Generator 6.2i*, which *Xilinx* is willing to donate to universities. After downloading *System Generator* from the *Xilinx* website, extract the file to a temporary directory, open Matlab, change the working directory to the temporary directory and write 'setup' on the command line.
- You can enable the use of the board throughout the network by right-clicking on the *Nallatech* icon in the *system tray* and selecting the 'Start TCPIP Server' option.
- The Reed-Solomon encoder/decoder cores are licensed separately fom *Xilinx.* If one intends to use these cores, they can be obtained via a donation from the *Xilinx University Program* as mentioned above. After obtaining the licences in the form of zip-files, these should be extracted to the root of the 'C:' drive (or the Windows home drive if not 'C:'). This will create a '.Xilinx' folder and automatically enable the use of the Reed-Solomon blocks within *SystemGenerator.* Note that the licence is fixed to the physical address (MAC address) of the network adapter of the computer for which the licence was acquired and therefore can be used only on that specific computer.
- Install our development platform located on the CD supplied with this thesis in the 'comm_library' folder. To install the library, copy the folder to a directory of your choice and set *Matlab* path ('File' > 'Set Path ...') to include the folder.
- The *Matlab* script 'constants.m' defines some constants that are used throughout the library as default values. For more convenience, review the script and execute it before using the library. Nonetheless, executing the script is not necessary.

You are now ready to use our developed platform. Open *Matlab* and use the command `open comm_library` to open the library with the platform modules. You can now use the modules to design your own algorithms and tests. Good luck!

Appendix B

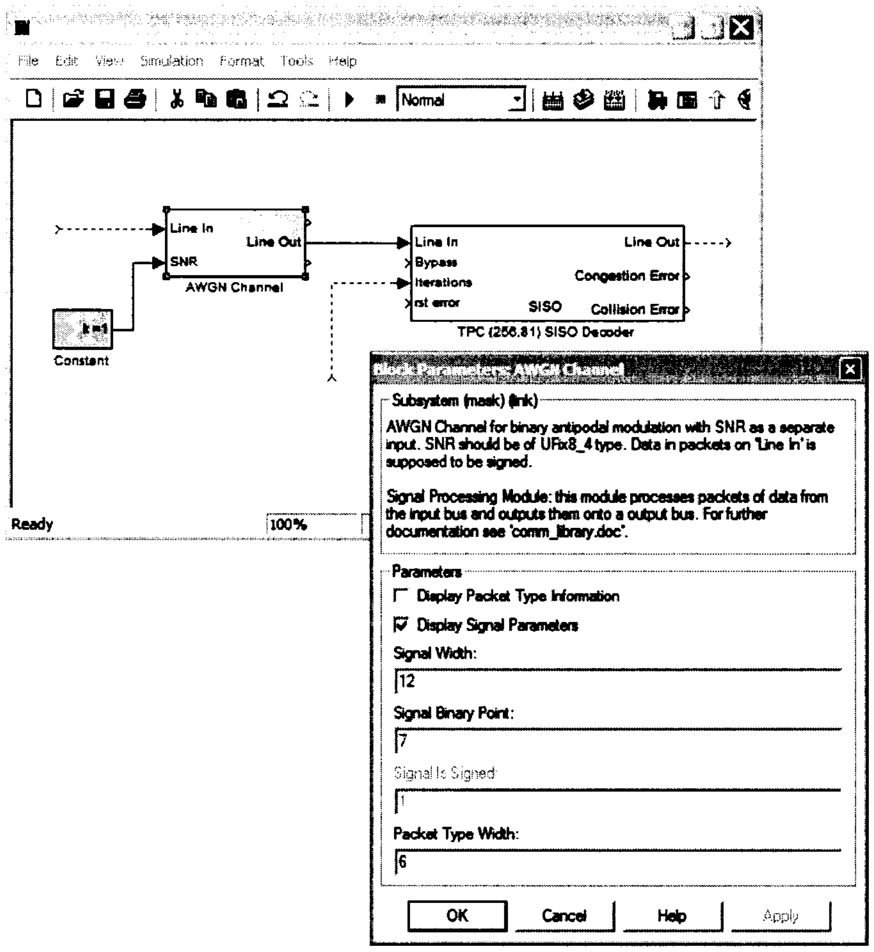
Library Reference Manual

In this appendix, we provide the technical documentation of our library for rapid prototyping of communication systems. In Section B.1, we briefly describe the interface of our developed library, which is the standard *Simulink* interface. Section B.2 includes the documentation of each individual library module. The structure of this section copies the organization of our library into *folder* containing related modules. Finally, Section B.2.11 serves as reference for development of modules. The section describes the internal parameters defined within a module, naming conventions and customizations of module's *mask* dialog box.

B.1 Using the Library Front-end

The library is a set of blocks implemented in *Simulink*. To use these blocks in a design, the user drags these blocks from a repository to his design window. In the design window, *Simulink* allows the user to place the blocks, connect them with signal lines and modify the blocks' parameters. Figure B.1 depicts the *Simulink* design window (background) together with an open *property* dialog (foreground) which displays the parameters of the *AWGN Channel* module. There are two ways how to move modules from the library into the user design:

- Open the library directly (the open communication library is depicted in Figure B.2) and use *drag and drop* or *copy/paste* functionality to move blocks from the library into his design,
- Open the *Simulink Library Browser* and pick a communication library block from the browser window as depicted in Figure B.3.



Block Parameters: AWGN Channel

Subsystem (mask) (link)

AWGN Channel for binary antipodal modulation with SNR as a separate input. SNR should be of UFix8_4 type. Data in packets on 'Line In' is supposed to be signed.

Signal Processing Module: this module processes packets of data from the input bus and outputs them onto a output bus. For further documentation see 'comm_library.doc'.

Parameters

☐ Display Packet Type Information

☑ Display Signal Parameters

Signal Width: 12

Signal Binary Point: 7

Signal Is Signed: 1

Packet Type Width: 6

OK | Cancel | Help | Apply

Figure B.1: A Simulink design window

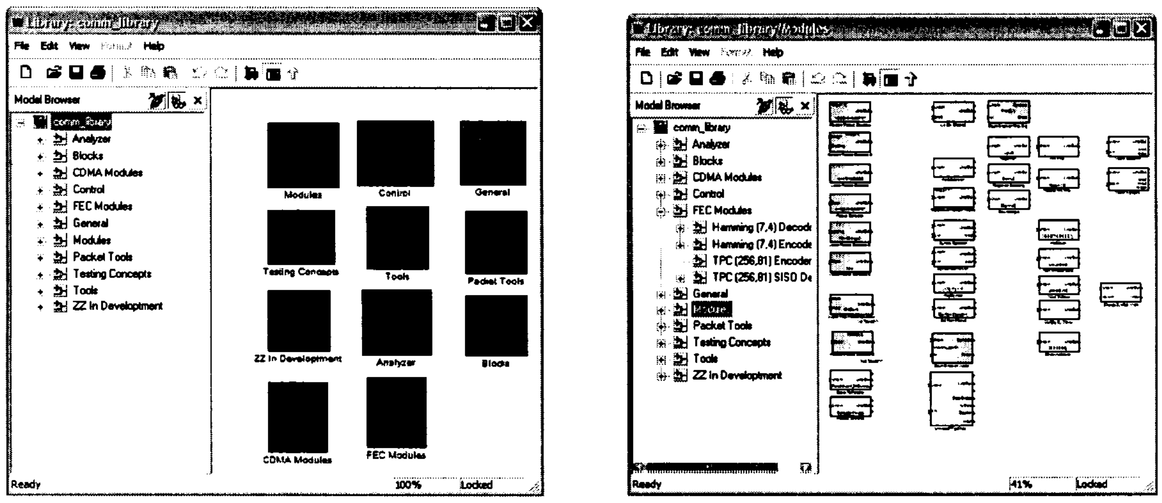
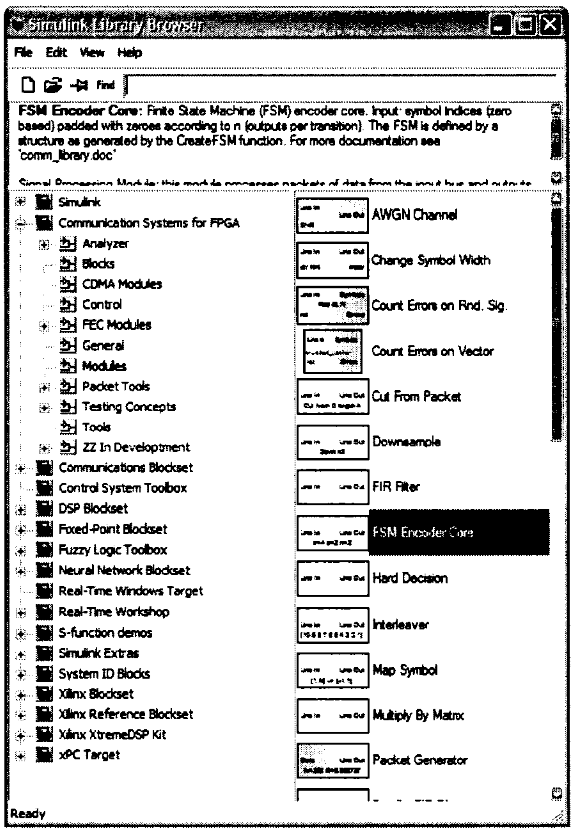


Figure B.2: An open communication library window



FSM Encoder Core: Finite State Machine (FSM) encoder core. Input: symbol indices (zero based) padded with zeroes according to n (outputs per transition). The FSM is defined by a structure as generated by the CreateFSM function. For more documentation see 'comm_library.doc'

Figure B.3: The *Simulink Library Browser*

B.2 Block Reference

The modules within the constructed library are organized into the following folders:

- Analyzer
- Blocks
- CDMA Modules
- Control
- FEC Modules
- General
- Modules
- Packet Tools
- Testing Concepts
- Tools
- ZZ In Development

These folders include *modules* that process packets and *blocks* that execute elementary operations on general signals, i.e., not necessarily on packets. The user will usually work with *modules*. These are located in the folders: *Modules*, *FEC Modules*, *Packet Tools* and *CDMA Modules*. The other folders contain

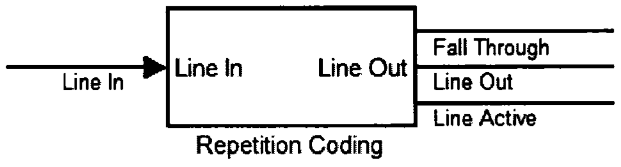


Figure B.4: An example of a module

blocks that are used internally in the library (*Control* folder and *General* folder), debugging *blocks* (*Analyzer* folder) and utility *blocks* (*Tools* Folder) that can be used to simplify certain design patterns. The remaining three folders (Blocks, *Testing Concepts* and *ZZ In Development*) were used during development for test and archival purposes and the content of these folders is undocumented.

In this reference manual, we provide the description of each of the *modules* and *blocks* in the library together with several examples of their use. A *module* is a *Simulink block* that processes packets and has an outward interface, as depicted in Figure B.4. For description of the internal structure of a module, please referrer to Chapter 3. You can find additional information on this subject in the documentation of the blocks contained within the *Control* folder as well as in the Module Specifications section.

B.2.1 Analyzer Folder

The data display and analysis blocks enable the user to view the data transmitted over a line during a simulation and store the data into the *Matlab* workspace for later analysis. These blocks are contained within the Analyzer folder which is depicted in Figure B.5 and in the following text; we give the detailed documentation of these blocks.

Start Logging

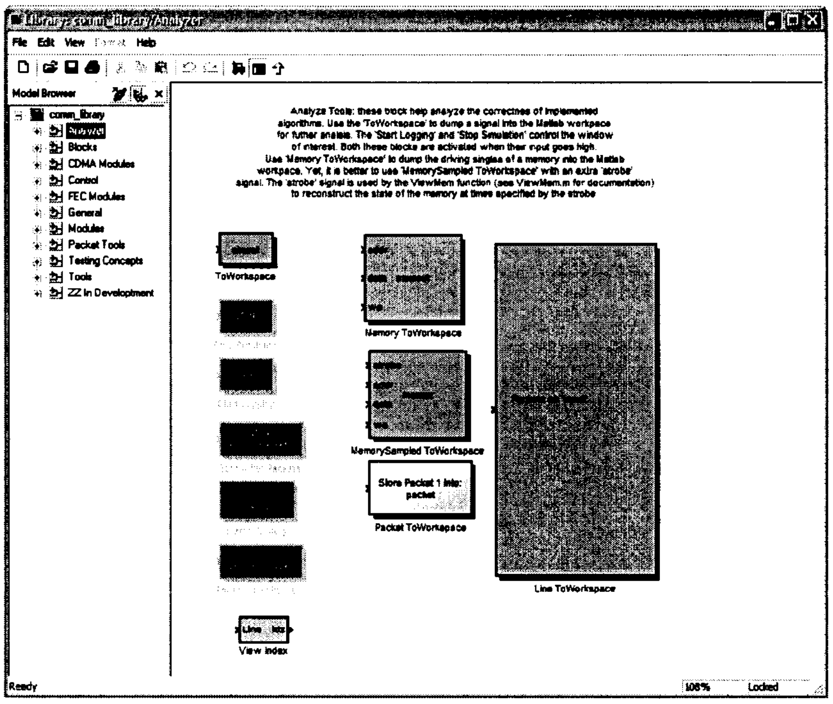
If the input becomes nonzero the block sends a start signal to all data logging blocks (do not combine with 'Enable Logging' block).

Start Immediately

Starts logging signals immediately after beginning the simulation.

Stop Simulation

Stops simulation when input is non-zero.



Analyze Tools: these block help analyze the correctnes of implemented algorithms. Use the 'ToWorkspace' to dump a signal into the Matlab workpace for futher analais. The 'Start Logging' and 'Stop Simulation' control the window of interest. Both these blocks are activated when their input goes high.
Use 'Memory ToWorkspace' to dump the driving singles of a memory into the Matlab workpace. Yet, it is better to use 'MemorySampled ToWorkspace' with an extra 'strobe' signal. The 'strobe' signal is used by the ViewMem function (see ViewMem.m for documentation) to reconstruct the state of the memory at times specified by the strobe

Figure B.5: Analyzer blocks

Stop After Packets

Stop simulation after a specified number of packets has appeared on the input line.

Pause After Packets

Pause simulation after a specified number of packets appeared on the input line.

ToWorkspace

Writes the values of the input into a Matlab variable (data are available only after the simulation is finished).

Packet ToWorkspace

Stores the content of the packet specified by the 'Choose Packet Number' parameter in the Matlab variable specified in the parameter 'Matlab Variable'.

Memory ToWorkspace

Writes the driving signals of a RAM block into a Matlab variable (data is available only after the simulation is finished).

MemorySampled ToWorkspace

Writes the driving signals of a RAM block into a Matlab variable (data is available only after the simulation is finished).

The extra 'strobe' signal is used by the ViewMem function (see ViewMem.m for documentation) to reconstruct the state of the RAM at times of interest. The function can store the state of the RAM either whenever the 'strobe' signal is high or when the 'strobe' signal changes level. ViewMem writes the states of the RAM at different times into a matrix.

Line ToWorkspace

Writes the values of the input into a Matlab variable (data is available only after the simulation is finished).

View Index

Outputs the *index* parameter of the last packet on the 'Line' input of the block.

Example of Using Analyzer Blocks

Figure B.6 and Figure B.7 show the usage of the Analyzer blocks to view packet data during simulation. *Header Spy* modules are used to display the packet parameters. In Figure B.6, *Packet ToWorkspace* blocks are used to store the content of the first packet in a *Matlab* variable (in this case *pgen* and *penc*). In Figure B.7, *View Index* blocks are used to monitor the individual packets, as these are processed by the design and *Line ToWorkspace* blocks are used to view the content of packets after the simulation is finished. A *Start Logging* block is included in the design to start logging the output into the *Line ToWorkspace* blocks when the first packet is output by the decoder.

The input parameters (in the *Simulink Constant* blocks) of the simulation can be modified on-the-fly during the simulation. This is useful to monitor the response of the system to these inputs.

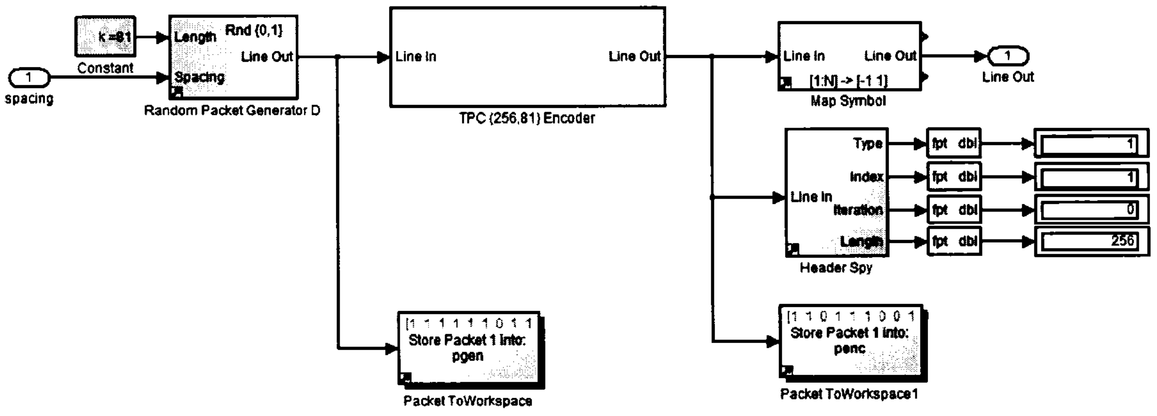


Figure B.6: Usage of *Packet ToWorkspace* and *Header Spy* blocks

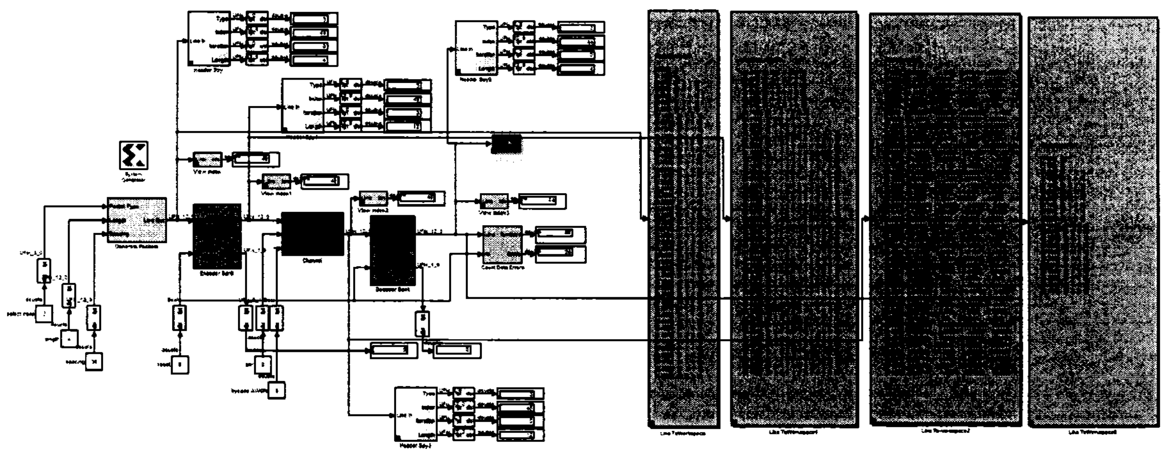


Figure B.7: Usage of Analyzer blocks for debugging

B.2.2 Blocks Folder

The content of this folder is undocumented. The folder was used during development, but the blocks contained within it do not have any functionality, neither are they used by other blocks in the library.

B.2.3 CDMA Modules Folder

This folder contains utility modules that can be used in the implementation of CDMA systems. The following text is the detailed documentation of these modules.

Apply Spreading

[1 0 0 1] --> [5 6 7|0 0 0|0 0 0|5 6 7]

Spreads input bits over multiple symbols. This module accepts packets of two types. One (Signature Packet Type) sets the spreading signature and the other (Packet Type) is the data packet type.

The data packet must be a binary packet containing only '0' and '1' symbols. The spreading sequence can contain arbitrary symbols, i.e., non-binary spreading is possible. The number of CDMA chips per input bit is equal to the length of the spreading sequence which is equal to the length of the supplied signature packet.

Bit Delay

Introduces a delay to the packet where the delay is in bits and therefore, this operation will alter symbols within the packet by shifting bits within a symbol and appending bits from a previous symbol.

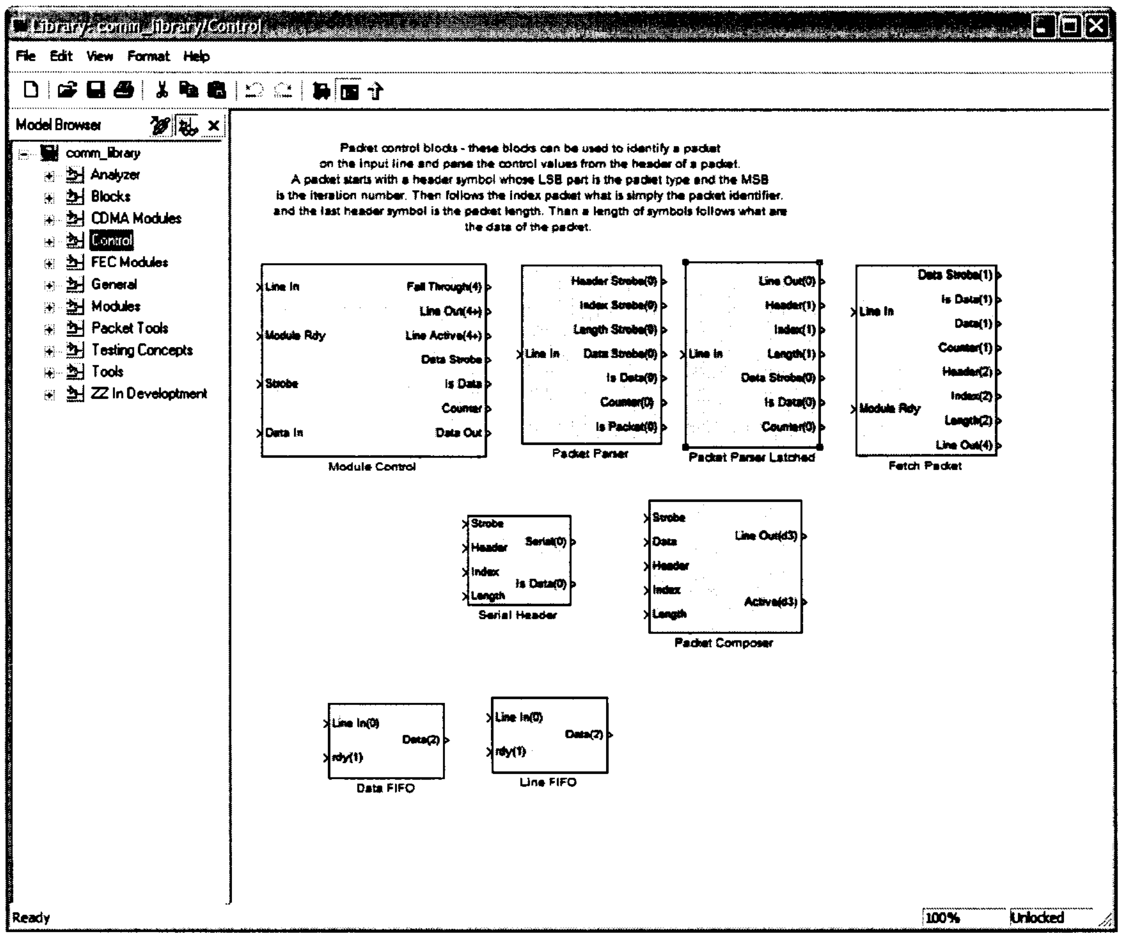
B.2.4 Control Folder

This folder contains elementary packet handling blocks. These blocks are used within all library modules to identify and retrieve packets from the input line as well as compose a packet on the output line. The contents of the folder is depicted in Figure B.8 and in the following text, we give the detailed documentation of these blocks.

Line FIFO

Line FIFO stores the packets on the line in a FIFO memory. If 'rdy' is high, packets are output on 'Line Out'. If 'rdy' becomes low in the middle of a packet, the output stream will be interrupted and will resume when 'rdy' becomes high again.

If the FIFO is empty the block has an input-output latency of 2. Note that the rdy-output latency is 1.



Packet control blocks - these blocks can be used to identify a packet on the input line and parse the control values from the header of a packet. A packet starts with a header symbol whose LSB part is the packet type and the MSB is the iteration number. Then follows the index packet what is simply the packet identifier. and the last header symbol is the packet length. Than a length of symbols follows what are the data of the packet.

Figure B.8: Control blocks

Data FIFO

Data FIFO stores data from packets on the line in a FIFO memory. If 'rdy' is high then data is output on 'Line Out'. If 'rdy' becomes low in the middle of a packet, the output stream will be interrupted and will resume when 'rdy' goes high again.

If the FIFO is empty, the block has an input-output latency of 2. Note that the rdy-output latency is 1.

Packet Composer

On strobe marking the beginning of the data on the data input this block outputs serially: the header, the index, the length and afterwards length symbols from the data channel. Naturally, it has a latency of 3 on actual data, but the header symbol is output with zero latency.

Serial Header

On a pulse on the strobe input the block outputs the header, index and the length serially. Header is outputted with zero latency.

Fetch Packet

Used to fetch a packet from the input line that has a correct type. If the packet type does not correspond to the 'Select Packet Type' parameter of this block (use zero in this parameter to select all packets) or if the connected module is busy (the 'Module Rdy' input is low) then the packet is sent to the 'Line Out' and not to the connected module at 'Data Out'.

Packet Parser

Packet Parser creates control signals from a packeted input line. No latency.

Packet Parser Latched

Packet Parser block with latches on the outputs (see Packet Parser). Packet header outputs become valid with latency 1.

Module Control

Module Controller block parses the necessary signals from the input line and if the 'Module Rdy' is high and a packet arrives that fits the packet type number then it is sent to the bock. It is the responsibility of the connected module to emit the data ready 'Strobe'. The size of the data block can be multiplied by a power of 2 with the 'Length Shift' parameter. The data in/out are forced to signed arithmetic with mask defined precision.

Latency: 4 clock cycles from 'Line In' to 'Line Out' plus the latency of the module.

Use of the Control Blocks

The *Fetch Packet* and *Packet Composer* blocks can be used to implement a *module* that conforms to our *module design specifications.* The simplest scenario of using these modules is demonstrated in Figure B.9. In this case, the *Fetch*

Packet block retrieves the packet from the 'Line In' input of the module and the parsed packet parameters are directly forwarded to the *Packet Composer* block, which will reconstruct the packet at the 'Line Out'. The only parameter that is modified is the length of the packet. This parameter is set to a constant that is specified in a module parameter. Therefore, this module will *cut* the packet down (or up) to a specified length.

Most modules are implemented using *Fetch Packet* on the input side and *Packet Composer* on the output with the module logic implemented in between. Nonetheless, the *Fetch Packet* block can be replaced with the *Store Packet* block (as demonstrated in Figure B.10) and the *Packet Composer* block can be replaced with the *Send Packet* block. These blocks provide a memory interface that allows analyzing or constructing the involved packet in memory.

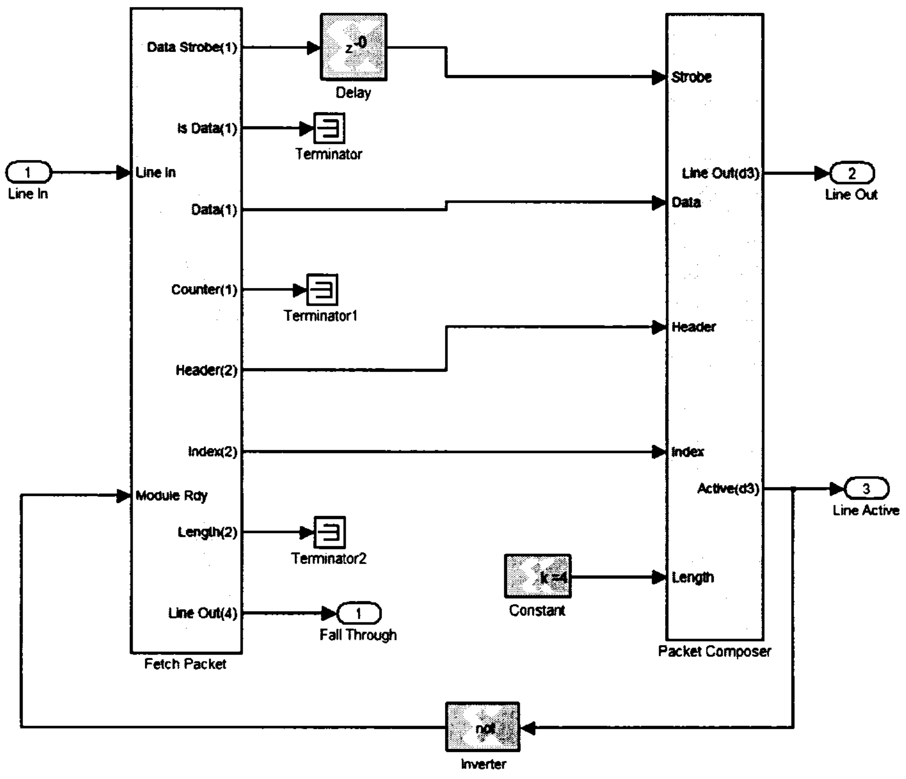


Figure B.9: Schematics of the *Cut From Packet* module

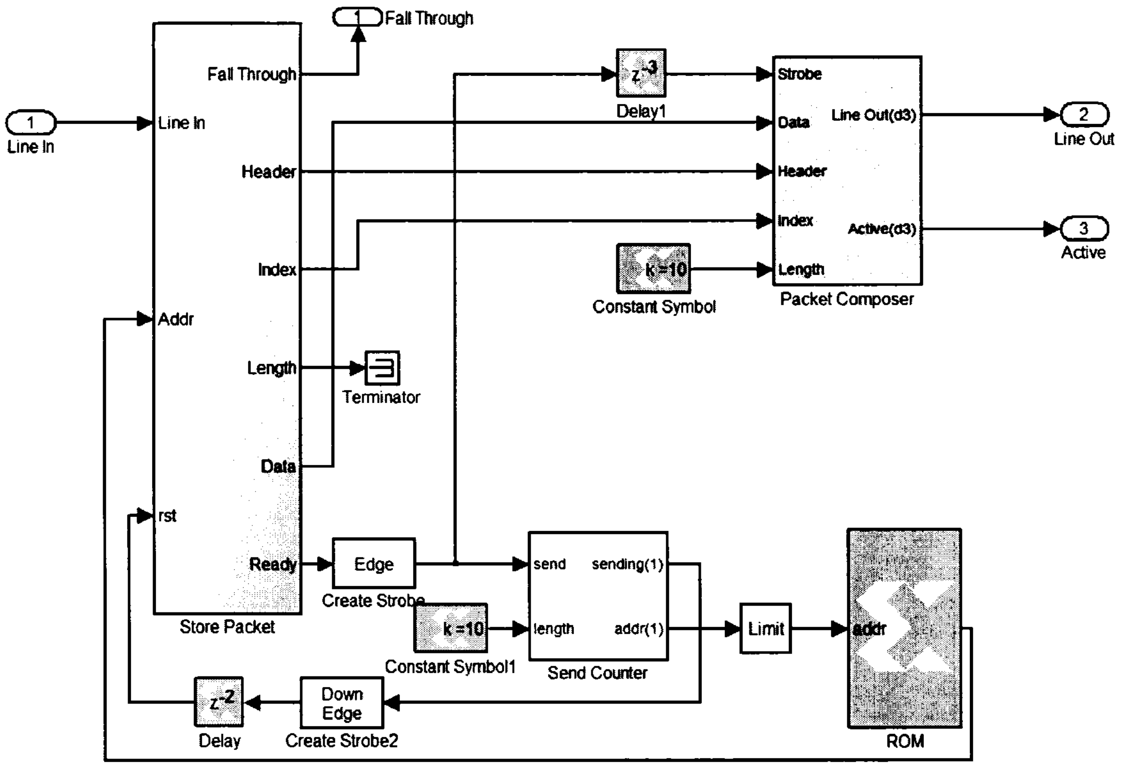


Figure B.10: Schematics of the *Interleaver* module

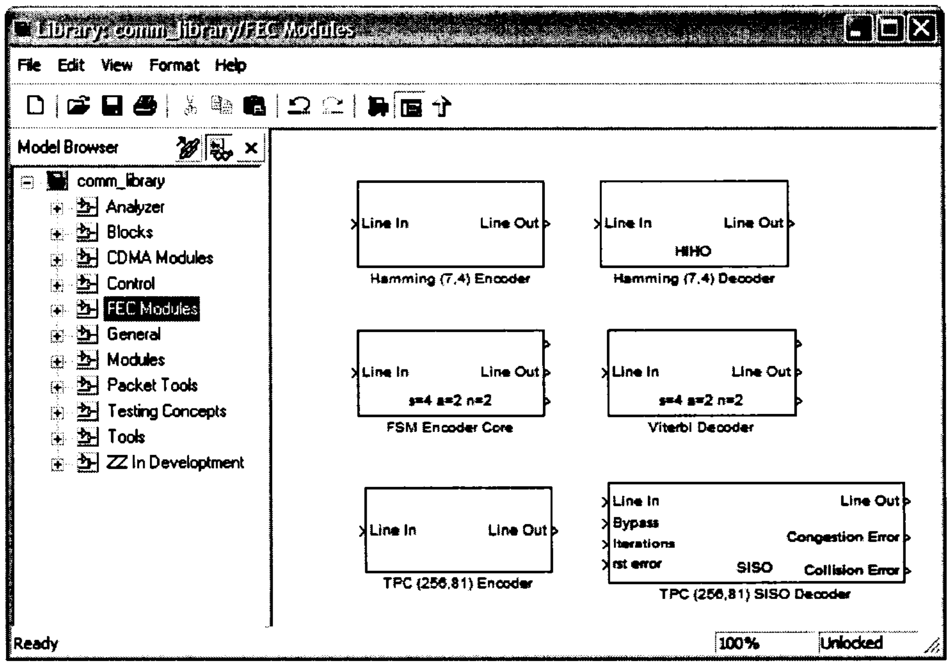


Figure B.11: Forward error correction modules

B.2.5 FEC Modules Folder

This folder includes the error-control modules implemented as an example of the usage of the library. This includes the Hamming (7,4) code, the *Viterbi* decoder with an *FSM Encoder* for a convolutional code and a $(4,3)^4$ Turbo Product Code. The content of the folder is depicted in Figure B.11 and, in the following text, we give the detailed documentation of the error-control blocks.

TPC (256,81) Encoder

Encodes an 81 bit packet of unsigned binary symbols (zero and one) into a packet of 256 signed symbols -1 and +1 with the given binary point.

TPC (256,81) SISO Decoder

This module implements iterative decoding of a (4,3)^4 product code. The code input can be represented by a four-dimensional hypercube with 3 bits in each dimension and a (4,3) parity applied on these three bits to generate a four-dimensional hypercube with 4 bits in every dimmension. The module is able to process 6 decoding iterations (an iteration includes decoding along all four dimensions) with a continuous stream of packets at the input. If more iterations are desired, the packets have to be spaced. If the module is not able to decode the packets at the given rate, it will output a 'Congestion Error'. The 'Collision Error' is just an internal check and the value of this output should always stay zero.

The packet values should be 3 bit probability estimates that the symbol '1' was transmitted (give what was detected in the receiver). This estimation is not part of this module. The output of this module are 3 bit probabilities of the sent symbols being '1' given the received information and given that the message was encoded using the (4,3)^4 code.

FSM Encoder Core

Finite State Machine (FSM) encoder core. Input: symbol indices (zero based) padded with zeroes according to n (outputs per transition). The FSM is defined by a structure, as generated by the CreateFSM function. This module is just the core of the encoder, thus certain assumption must hold and certain pre-processing of the input packets must be done:

- Input line - the input line should contain indices (zero based) of the symbols from the input alphabet. These must be padded with zeroes for the input and output packet lengths to match (because this module is on purpose implemented without storing the whole packet in memory).
- Input line - the FSM does a transition on every single input on the input line. If transition on multiple input symbols is desired it is necessary to map these multiple symbols into one symbol first.
- Output - at least 2 outputs per transition need to be generated. This is due to the implementation of the encoder (one of these outputs can be later punctured to generate only one output per multiple input symbols).

The encoder is defined by a Transitions and Outputs matrix. The Transitions matrix is a states-by-symbols matrix that gives the new state after the transition. The Outputs matrix is a states by n*symbol matrix that gives the n outputs on a transition. The FSM structure, used for this module, contains additional computed parameters. The CreateFSM function computes these parameters from the Transitions and Outputs matrix. The FSM structure contains the following parameters:

```
% FSM structure defines a Finite State Machine:
%
% fsm.Transitions    - states by symbols matrix of new state
% fsm.Outputs        - states by n*symbols matrix of output symbols
% fsm.nStates        - number of states (redundant)
% fsm.nInSymbols     - size of input alphabet (redundant)
% fsm.nOutputs       - number of outputs per transition (redundant)
% fsm.nOutSymbols    - size of output alphabet (redundant)
```

The nOutSymbols parameter is simply the maximal symbol plus one. Thus is the output contains zero based indices than this parameter is the size of the output alphabet. But the outputs do not have to be zero based symbol indices than this parameter does not make sense.

This is an example of an FSM with 4 states and 2 outputs per transition (if FSM is in zero state, on zero it transits to 0 and outputs 0 1, while on one it transits to 1 and outputs 1 1):

```
CreateFSM([0 1; 3 2; 1 0; 2 3], [0 1 1 1; 0 0 1 0; 1 1 0 0; 0 1 1 1])
```

Viterbi Decoder

This is a serial implementation of the Viterbi algorithm. The module can decode a code encoded by any state machine. The state machine used must be supplied to this module as a parameter. For the description and structure of this parameter, see the 'FSM Encoder Core' module.

The algorithm finds the "shortest path" though the trellis of the code. The metric assigning lengths to the edges of the trellis is implemented within the module in the 'Symbol Measure' subsystem. Modify this subsystem to transform the received signal to a proper *additive* probability measure.

Signal Processing Module: this module processes packets of data from the input bus and outputs them onto an output bus. For further documentation, see 'comm_library.doc'.

Hamming (7,4) Encoder

This module encodes 4 symbols on the input using the Hamming (7,4) code by multiplying the codeword by the generator matrix G given below. The input symbols should be either zero or one; unsigned with binary point at zero. The symbols in the 7 symbol output packet are also zeroes and ones; unsigned with the binary point at zero.

```
HamG =
     1     0     0     0     1     1     1
     0     1     0     0     0     1     1
     0     0     1     0     1     0     1
     0     0     0     1     1     1     0
```

Hamming (7,4) Decoder

This module accepts a 7 symbol packet encoded using the Hamming (7,4) code and outputs the decoded 4 bits. All packets should contain only zeroes and ones; unsigned with binary point at zero. The output has the same format as the input.

B.2.6 General Folder

These blocks are used internally by other blocks and modules in the library. The blocks include several types of counters, loop controllers, random sequence generators and special purpose RAM and ROM blocks. Normally, the user will not utilize these blocks.

Inner Product

Computes the inner product of the data on the 'data' input with the vector specified as parameter. Data should be sent 2 clock cycles after the reset pulse. The 'data' vector is assumed to have the same size as the stored vector. The result is outputted onto the 'pipe'. In particular, the pipe is transparent, but in clock cycle 6+length(vector), the pipe symbol is replaced with the result.

FOR Cycle Controller IF

Variant of the FOR Cycle Controller with variable upper limit and a fixed length of the FOR Cycle. See 'FOR Cycle Controller' block documentation for details on this block.

FOR Cycle Controller I

Variant of the FOR Cycle Controller with variable upper limit of the FOR Cycle. See 'FOR Cycle Controller' block documentation for details on this block.

FOR Cycle Controller F

Variant of the FOR Cycle Controller with fixed time for one step of the FOR Cycle. See 'FOR Cycle Controller' block documentation for details on this block.

FOR Cycle Controller

After reset the i output goes from the Initial Value to the Final Value with the specified Step. The Finished output goes high one clock cycle before the for cycle finishes. The connected circuitry within the for cycle is responsible for driving the step input. The step input must go high Delay+1 clock cycles before the circuitry finishes processing cycle with index i and i should be increased to i+Step. sub rst goes high one clock cycle before i increases. Use this reset signal to initialize the circuitry within the cycle.

(Why the "goes high one clock cycle before"? ... to avoid wasting clock cycles for initialization)

Select RAM

Allows switching of inputs between two banks of RAM of the same size using a select signal. If select is 0 than addr0, data0 and we0 are routed to bank 0. If the select signal is 1 than these signals are routed to bank 1. The addr1, data1, we1 signals are routed in the opposite way. The outputs are also routed this way.

FlipFlop

Flips the output whenever the flip input is high.

Send Counter

Address counter which starts counting up when there is a strobe on the 'send' input. The address width is equal to ceil(log2(MaxLength)).

Setable Slow Dwncnt (block name misspelled in the library)

Down-counter with divided rate. The counting starts at 'value' and the output at the 'Slow Counter' decreases every N clock cycles. The sub-counter counts from 0 to N-1 for every 'Slow Counter' level. The 'Running' output is high till the counter counts to zero (including zero). The block has a latency of 1.

Note: Using saturation in subtraction block (maybe that it can be implemented more effectively).

Down Edge

Fires a pulse when the input goes down.

Limit

This block should be used at the address input of a memory block to limit the index to only the valid values.

Note: If you use packet sizes of a power of two you can simplify this block by just leaving the slice block there.

Random Sequence Generator

This module generates a random binary sequence. Use the 'Seed' input together with the 'Load' input high to initialize the random number generator to a certain value. This block is used for random packet generation as well as for verification of transmitted packets generated using the random packet generation (implemented using a 32 bit linear feedback shift register).

Slow Downcounter

Down-counter with divided rate. The counting starts at 'value' and the output at the 'Slow Counter' decreases every N clock cycles. The sub-counter counts from 0 to N-1 for every 'Slow Counter' level. The 'Running' output is high till the counter counts to zero (including zero). The block has a latency of 1.

!! The maximum division ratio is 2^16 as the sub-counter has 16 bits!!)

Note: Using saturation in subtraction block (maybe that it can be implemented more effectively).

Setable Downcounter (block name misspelled in the library)

Down-counter. Set 'set' input to high and 'value' input to the number of count-steps. The counter will count from value-1 to 0. 'Running' is high while the counter is running. Counter has a latency of 1.

Notes: This can be implemented more effectively. Using saturation on subtraction - that costs extra resources.

SR Latch

Set-reset latch implemented using a flip-flop.

Notes: Probably not implemented optimally.

Edge

Emits a pulse when the input goes high.

Memory Feed

This block stores the specified vector in memory and then feeds it to the output. When the last value of the vector is reached, the block starts feeding the data from the beginning again.

Memory Feed (Sgn)

This block stores the specified vector in memory and then feeds it to the output. Wwhen the last value of the vector is reached, the block starts feeding the data from the beginning again. As opposed to the 'Memory Feed' block, this block sends the given vector as signed values.

B.2.7 Modules Folder

Data Conversion and Simple Operations Modules

The elementary Packet/Data handling modules enable elementary operations on data contained within a packet. These modules, such as *Symbol Operation* and *Symbol Conversion,* that enable us to execute operations on all data symbols within a packet. Additional modules execute specific symbol operations.

Symbol Operation Module

This module connects to external user circuitry which implements an operation on one symbol of the packet. The *Symbol Operation* uses this circuitry to execute the operation on every symbol in a packet. The module can be used to implement operations such as adding three to all symbols, taking all symbols modulo 4 or masking symbol bits using a custom mask.

Symbol Conversion Module

The *Symbol Conversion* module is similar to the *Symbol Operation* module with external user circuitry; executing the conversion of one symbol. On top of the *Symbol Operation* module, this module enables to alter the format of a packet, i.e., to change the bit width of a symbol and the bit width of the *Packet Type* field within the packet header.

Map Symbol Module

The *Map Symbol* module maps symbols representing integer values between *zero* and *N* to a user defined symbol. The module is parameterized using a vector of length *N+1* that defines the values to which the symbols are mapped.

Hard Decision Module

This module implements a hard decision on a soft input represented by a signed fixed point number. All negative symbols are mapped to *-1* and all positive symbols are mapped to *+1*.

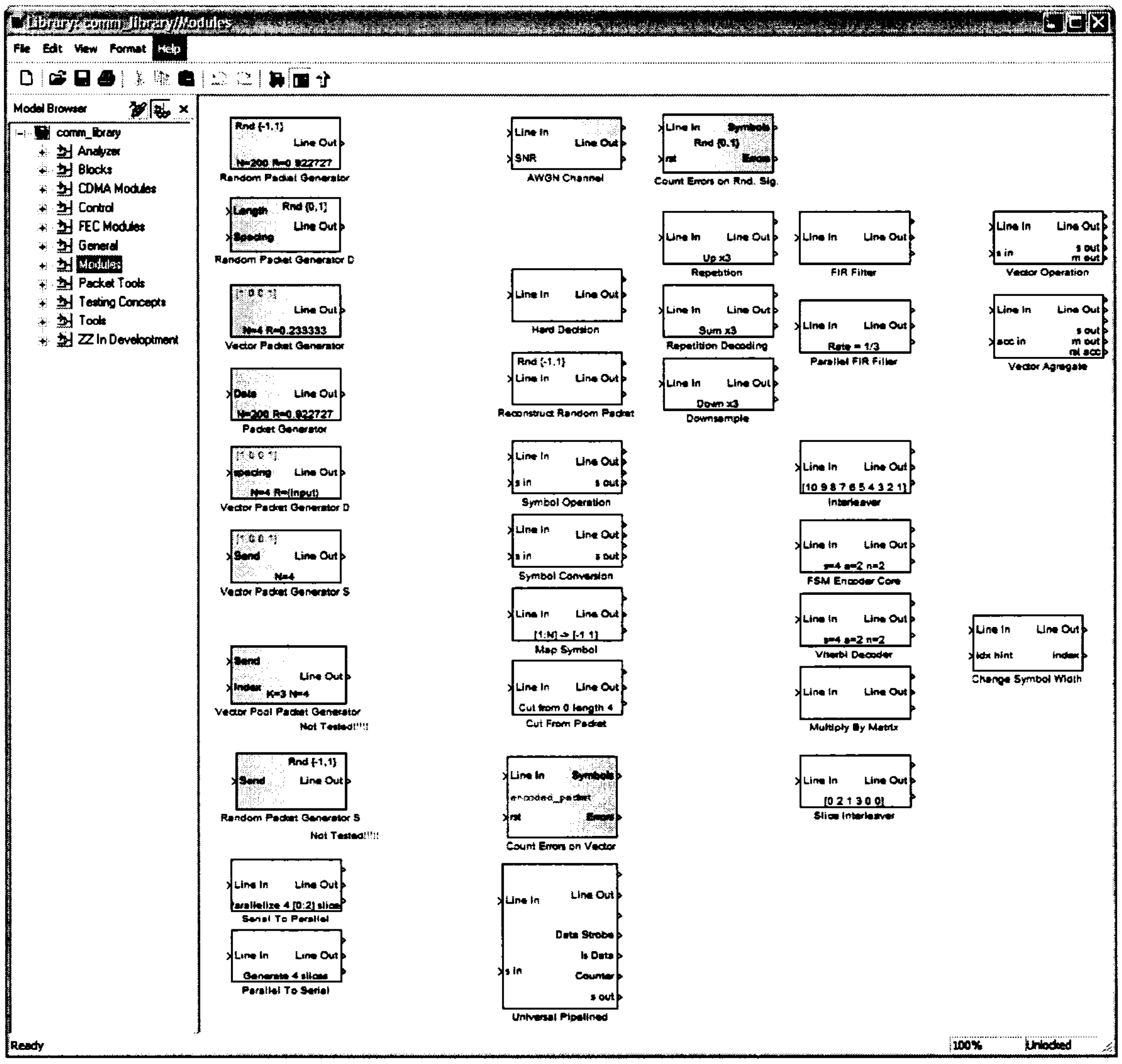


Figure B.12: An open communication library window

Change Symbol Width

This module changes the symbol width. The extra input and output signals take care of repairing the 'index' parameter of the packet after scaling down the symbol width to a smaller number and later scaling it up.

When resizing the symbol down, the module outputs the 'index' of the packet. This 'index' output can be used as the 'idx hint' (index hint) input of another module that will increase the width of the packet symbols back to the original. This is done to repair 'index' if the value has overflown due to low number of bits in the symbol while scaled down. When decreasing the width of symbols the 'idx hint' input should be a single bit constant equal to zero. When increasing the width of a symbol, the 'idx hint' should be an index of a packet that entered the system later than the current packet and the width of this input should equal to the requested width of the symbol at the output of such block.

Vector Operations and Manipulation Modules

These modules interpret the packets as data vectors and execute operations on these vectors. Modules accept packets of arbitrary length dynamically adjusting the operation to the length of the packet. Nevertheless, most modules require the maximum packet length to be defined as a parameter.

Multiply By Matrix Module

This module interprets the packet as a row vector and multiplies it from right by a matrix specified in a parameter. The output is also a row vector that is used to construct the output packet. The operation of this module is parallelized using the *Generate* statement. Therefore, no spacing of packets at the input is necessary to allow for processing time of the module. Packets can be processed at the speed limited only by the full use of the input and output lines.

Interleaver Module

This module permutes (interleaves), repeats or punctures the symbols within a module. An arbitrary sequence of numbers from 0 to L-1, where L is the length of the input packet, is given to the module as a parameter. Each number in the sequence is interpreted as a zero based index into the input packet and an output packet (of the same length as the given sequence) is formed from symbols in the input packet as these indices.

Example: Using sequence [2 2 0 4] on input packet [1 2 3 4 5 6] will result in output packet [3 3 1 5].

Slice Interleaver Module

This module is similar to the Interleaver module. It interleaves (punctuates, repeats) the input packet according to a vector. The vector defines the indices of the data from the input packet (zero based!!!) that should be presented at the output. Different from the Interleaver module - this module considers the input data to consist of slices of K bits. Thus, if the input packet consists of N words of L bits, the input will be considered to be a vector of length N*L.

Example: If the module receives a packet consisting of 2 six-bit data words [110001 101100] and it is asked to consider the input data in slices of 3bit than if the given "permutation" vector is [0 2 1 3 0 0] it will output a packet of 3bit words of length 6: [001 100 110 101 001 001].

FIR Filter Module

This module applies a finite impulse response filter to the packet at the input. The impulse response is defined in a parameter of this module. This changes the length of the packet. If the input packet was *n* symbols long and the impulse response has *k* symbols, the output packet will have $n+k-1$ symbols. The operation can be expressed as a convolutional sum:

$$b_i = \sum_{j=1}^{K} a_j \cdot h_{i-j}$$

Vector ***a*** represents the input packet, vector ***b*** the output packet and ***h*** is a vector of length *k*, which represents the impulse response of the filter.

The module can be directly used as a convolutional code encoder. Nonetheless, this is a serial implementation of the filter. Therefore, the module will be busy for $(n+k-1)\cdot k$ clock cycles after accepting a packet.

Parallel FIR Filter Module

The *Parallel FIR Filter* executes multiple filtering operations on a specific packet in parallel. Given an *r* by *k* matrix **H**, the module interprets each row of the matrix as an impulse response and convolves the response with the input packet. The resulting outputs are then merged into an output packet by first taking the first symbol from all results, then the second symbol and so on. Therefore, if *n* is the length of the input packet, the output packet will have $(n+k-1)\cdot r$ symbols. We can say the module is a rate *1/r* convolutional encoder over real field.

The multiple filtering operations are executed in parallel. Therefore, the module stays busy for the same amount of time as the *FIR Filter* module; $(n+k-1)\cdot k$ clock cycles after accepting a packet.

Vector Operation Module

This module enables the user to execute a custom operation using two vectors. The module accepts packets of two types (as specified by the 'Packet Type' parameter within the packet), A-type and B-type. Load the block with a B-type packet, then any A-type packet can be processed with the information from the B-type packet cyclically wrapped (B packet data is presented on the 'm out' and A packet is presented on the 's out' ... 's in' should be computed).

Set the Latency parameter of the block to the latency of the operation (thus the latency from 's out' to 's in').

Vector Aggregate Module

The *Vector Aggregate* Module can be used to execute custom operation on one vector while using information from another vector. The module accepts packets of two types (as specified by the 'Packet Type' parameter within the packet), A-type and B-type. Load the block with a B-type packet, then any A-type packet will be processed by vectors of the length of the B-type packet. This block can be used to implement a matrix multiplication by simply connecting a MAC to the s, m, rst outputs and the acc input. As opposed to the vector operation module, this module will output only one symbol per L input symbols, where K is the length of the B-type packet.

Set the Latency parameter of the block to the latency of the operation (thus the latency from 's out' to 's in').

Packet Form Editing Modules

The packet-form editing modules enable us to alter the structure of a packet by cutting symbols out of a packet, repeating symbols or aggregating symbols into symbols of increased bit width.

Cut From Packet Module

This module enables us to cut a block of symbols out of the middle of the packet and transmit this block of data as a new packet. The module takes two parameters: the index of the first symbol of the block and the length of the block.

Serial To Parallel Module

The *Serial To Parallel* module enables to speed up processing of data by parallelizing symbols. This is done by increasing the bit width of a symbol by a factor k, where k is a user defined parameter, and merging consecutive groups of k symbols into one "wider" symbol. This operation will shorten the packet by a factor of k. The *Parallel To Serial* module has a reverse effect.

Parallel To Serial Module

This is the reverse of the *Serial to Parallel* module. Symbols of bit with $k \cdot n$ are cut into k sub-symbols of width n, where k is a user defined constant and n can be derived from the original symbol bit width. These sub-symbols are then transmitted sequentially in a packet with its length increased by a factor of k.

Repetition Module

This module repeats each symbol k times in the output packet. This increases the length of the packet by a factor of k. The constant k is specified as a parameter of the module.

Downsample Module

The *Downsample* Module reduces the length of a packet by a factor of k. This is done by sampling only the first symbol from each group of k symbols in the input packet.

Communication System Simulation Modules

The communication system simulation modules facilitate the evaluation of communication systems, e.g., by providing random packet generators, channel simulation blocks and error counting modules. By including these modules in a setup and translating this setup into an FPGA, bit-error-rate simulations can be executed at full hardware speed.

Random Packet Generator Modules

The *Random Packet Generator* modules facilitate the generation of binary pseudo-random packets. The generation of pseudo-random numbers is implemented using a 32 bit linear feedback shift register. The generated packets are numbered sequentially and the packet number is stored in the *packet index* symbol inside the packet header. This *packet index* is then used as the seed for the random number generator that is used to generate the packet. Therefore, the content of a random packet can be reconstructed using the *packet index*. This enables easy checking whether a packet was altered during transmission. In fact, the *Count Errors on Random Packet* Module compares a packet to a packet reconstructed from the *packet index*, counts the number of positions where these two packets differ and declares this as the number of errors.

There are two variants of this module. One requires that the length and the spacing of the packets are defined as constants at design time, whereas the other accepts these parameters as inputs that can dynamically change during execution.

Vector Packet Generator and Packet Generator Modules

Several variants of a random packet generator module are available in the library. These modules generate packets containing a random sequence of *zero* and *one* symbols. The value of the *zero* and *one* symbols can be defined in the modules' mask. Therefore, it is possible to generate packets containing a random sequence consisting of *+1* and *-1* as well as *0* and *1*.

AWGN Channel Module

The AWGN Channel module adds Gaussian noise to packet symbols representing fixed-point signed numbers. The module assumes a modulation scheme with a unit average energy was used (such as BPSK to +1 and -1). The additional SNR input of the module controls the signal to noise ratio. This input is an 8 bit fixed point number with 4 fractional bits. Therefore, only values between 0dB and 15.94dB in increments of 1/16dB are possible for this input.

Count Errors on Random Packet Module

This module counts the errors that occurred in the transmission of a packet. The module assumes that the packet was generated using the *Generate Random Packet* module. This module generated a deterministically random packet depending on the seed stored in the *Index* parameter stored in the packet header. The *Count Errors on Random Packet* module uses the *Index* parameter to generate the original packet. By comparing the newly generated random packet and the received packet, the module determines the number of errors. This enables us to count the errors without storing the original packet.

Communication Systems Algorithms

The *FSM Encoder Core* and the *Viterbi Decoder* modules are included in the 'Modules' folder only to assure compatibility with some older designs that assume that these modules are located in the 'Modules' folder. Please refer to the 'FEC Modules' section of this reference for documentation of these modules.

FSM Encoder Module

Please refer to the 'FEC Modules' section for documentation of this module.

Viterbi Decoder Module

Please refer to the 'FEC Modules' section for documentation of this module.

B.2.8 Packet Tools Folder

This section of the library includes modules that enable to route packets between modules, process packets in an iterative loop, store packets and load them from memory.

Iterate Module

This module enables to iterate a packet or a number of packets in a loop. The module keeps a packet in a loop, while the *Iteration* parameter in the header of the packet is below a specified value. The *Iteration* parameter has to be incremented within the loop.

Line Cat Module

The *Line Cat* module joins two packet transmission lines into one. Packets on these two lines are synchronized using a FIFO buffer and a packet consisting of the concatenation of the symbols on both lines is presented at the output.

Line Collide Module

This module simply merges two lines into one. The merged lines can never be active at the same time. If they are, a collision occurs and the data on both lines will be lost.

Store Packet Module

The *Store Packet* module stores a packet from the input line into memory. The module then enables additional processing of the packet using its memory access interface.

Send Packet Module

This module enables the user to create a packet in memory and then to send it to an output line.

Header Spy Module

This module enables to analyze the header information of a packet on a line.

Packet Count Module

This module simply counts the number of packets transmitted over a specific line.

B.2.9 Testing Concepts Folder

The content of this folder is undocumented. The folder contains systems (schematics incorporating other modules from the library) used during the testing of library modules. The user can utilize these schematics as examples of how the respective modules should be employed.

B.2.10 Tools Folder

This folder contains blocks that can be considered as tools for hardware design and testing. The folder contains a bit-error-rate plot controller, a block that automatically drives the SNR input of a supplied system to test its performance over a range of values. It also contains the *Generate* module that allows the user to automatically generate regular logic structures.

BER Plot Controller

Control block to be used for the generation of BER plots. Apply a ramp on the ramp input and specify the step on the delta input.

Generate

The Generate block is used to generate multiple instances of the 'Element' block which is specified as a parameter in the mask (referenced by name). A copied and a hidden instances of it are created for every value of a vector given in the mask. The 'Element' block can contain configurable ports ('Goto' and 'From') that allow to cross-connect the generated elements.

For example, if you create an element that contains only an input ('From') port connected to an output ('Goto') port and you name the input port 'A[i]' and the output port 'A[i+1]', naturally the output port would be directly connected to the next block's input. Now you just need to create an 'A[1]' Goto port and an 'A[11]' from port to connect the generated elements to the surrounding design.

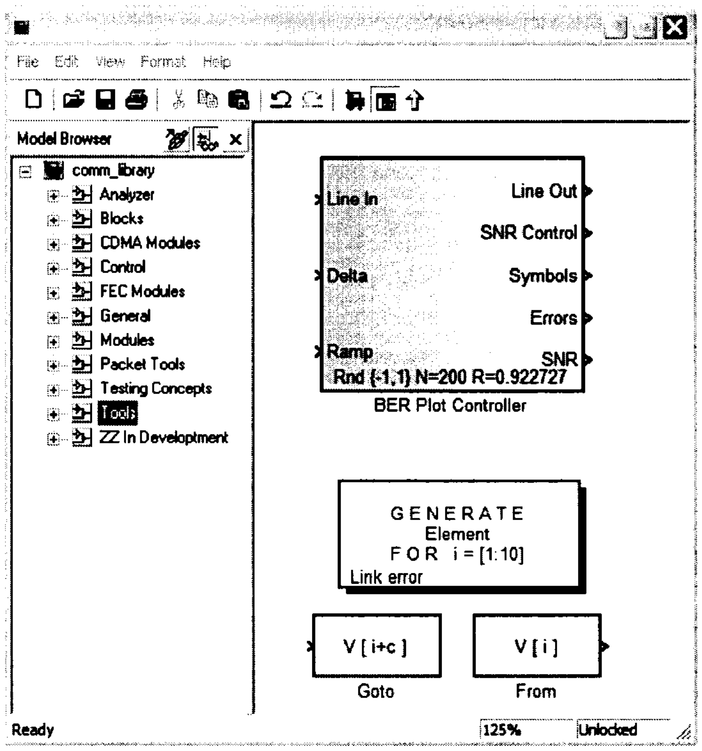


Figure B.13: An open communication library window

Use the 'Watch List' parameter to specify a list of numeric variables that should be watched for changes. If none of the specified variables changes the block will not regenerate the elements. This can be used to speed-up updates of complicated designs. Specify '[]' (empty matrix) to always update the block. Use a constant variable (i.e., '43') to specify that the block should never be updated. Use the global setting 'Force Update' to update all generate blocks.

Goto

The 'Goto' block defines an output port for use inside a generate structure. The port is specified by its name and index. Matlab expression using the iteration variable 'i' of the generate statement can be used to link the generated 'Element' blocks properly.

For example, defining an A[i] input port (From) and an A[i+1] output port (Goto) will chain the generated elements by the A port.

The 'Goto' block operates on the scope of a generate block. If there is no generate block in the same subsystem as the 'Goto' block, it refers to the innermost generate block in a possibly nested structure. If there is a generate block in the same subsystem then the 'Goto' refers to this generate block. As it is outside the generated elements, it cannot use the generation index 'i' - to overwrite this behaviour, check the 'Ignore scope' checkbox - then again the 'Goto' would reference a parental generate block.

From

The 'From' block defines an input port for use inside a generate structure. The port is specified by its name and index. Matlab expression using the iteration variable 'i' of the generate statement can be used to link the generated 'Element' blocks properly.

For example, defining an A[i] input port (From) and an A[i+1] output port (Goto) will chain the generated elements by the A port.

The 'From' block operates on the scope of a generate block. If there is no generate block in the same subsystem as the 'From' block, it refers to the innermost generate block in a possibly nested structure. If there is a generate block in the same subsystem then the 'From' refers to this generate block. As it is outside the generated elements, it cannot use the generation index 'i' - to overwrite this behaviour, check the 'Ignore scope' checkbox - then again the 'From' would reference a parental generate block.

B.2.11 ZZ in Development Folder

The content of this folder is undocumented. The folder contains development (or older) versions of modules in the library.

B.3 Module and Packet Specifications

All *modules* must have the same outward interface and the same outward behaviour. A *module* must conform to the following design specifications:

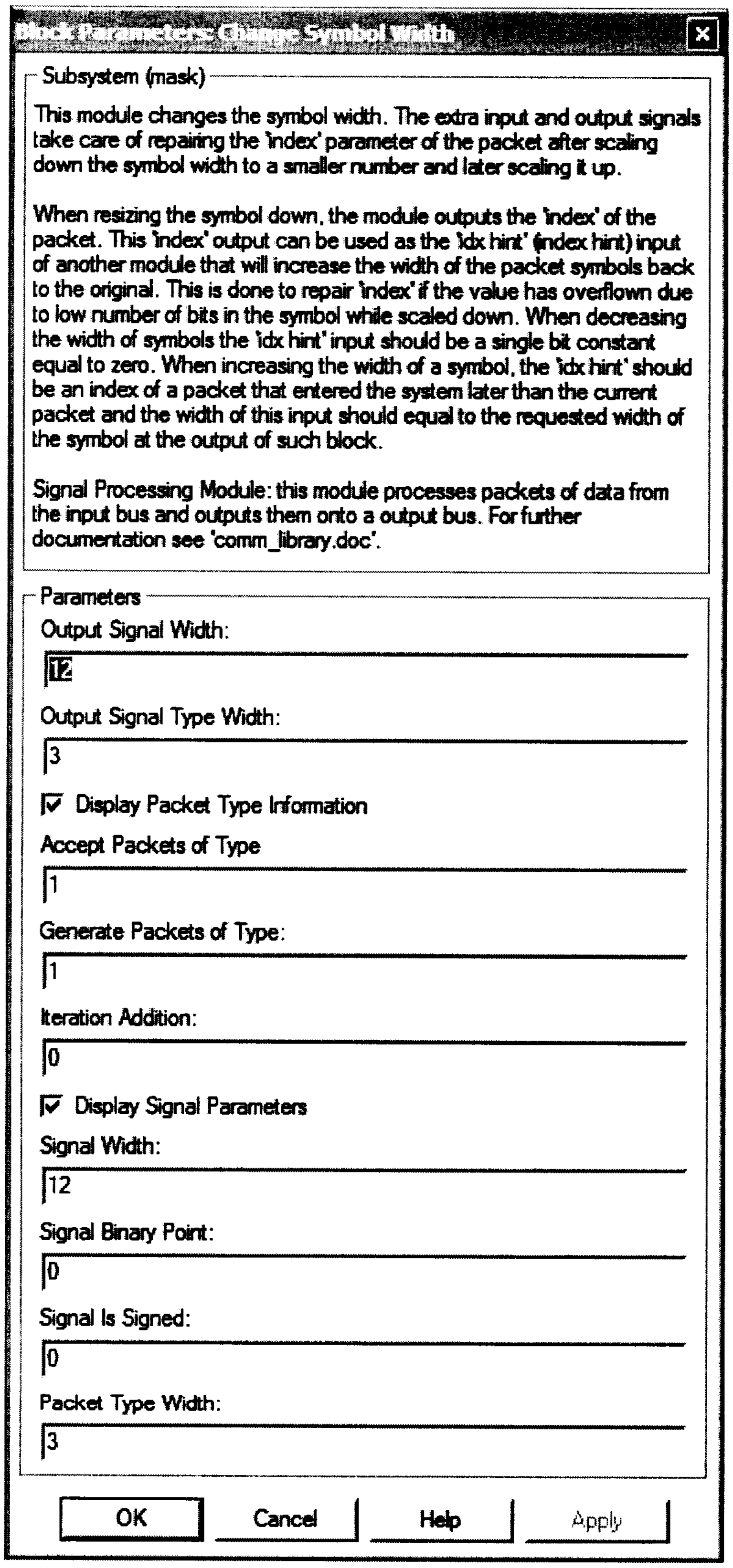
- The outward signal interface must be as depicted in Figure B.4. The module processes packets from the 'Line In' input and outputs these on the 'Line Out' if the packet was processed and on the 'Fall Through' output if it was not. The 'Line Active' output is high if there is data on the 'Line Out'.
- Additionally to module specific parameters, every module must have the parameters listed in Table B.2 and the parameters must be presented in the module mask (parameter dialog), as shown in Figure B.14.
- Packets processed by modules must have the structure described in Table B.1. A packet will be processed by a module only if the module is ready and the 'Packet Type' parameter of the packet is equal to the 'Accept Packets of Type' parameter of the module or if the latter parameter is equal to zero.

- The internal naming of the parameters specified in the mask should conform to the names given in Table B.2

The *Fetch Packet* and *Compose Packet* blocks implement the 'Line In' and 'Line Out' functionality that conforms to the specifications listed above. Please refer to the documentation of these two blocks. To implement a module, simply pass the parameters from the module mask to these blocks and implement the logic that will operate on the packet data. The design process of a module is described in greater detail in Chapter 3.

Table B.1: Packet Structure

Header	3 header symbols
Header Symbol Consist of packet type and iteration, where packet type occupies the bits from 0 to SignalTypeWidth-1 while iteration occupies the bits from SignalTypeWidth to SignalWidth.	The *header* symbol consists of two parts: the *packet type* and the *iteration*. The *packet type* is a number (user defined code) that should identify the meaning of the data. This information can be used by modules to selectively process packets (see Modules for further information). The *packet type* has to be nonzero. The *iteration* is a number that can be increased whenever a packet is processed by a module. Thus this number can be used if a packet should cycle through a module several times (see Modules for further information).
Index Symbol	The *index* symbol should be used for packet indexing (identification). Thus if the packets are out of order at the decoder output this number can be used to reorder them. But this symbol can be also used otherwise if necessary.
Data Length	The *data length* defines the length of the data portion of the packet. As this symbol has the same with as the line, the maximum data length is 2^(line width).
Data	The following *data length* symbols are the actual packet data.



Block Parameters: Change Symbol Width

Subsystem (mask)

This module changes the symbol width. The extra input and output signals take care of repairing the 'index' parameter of the packet after scaling down the symbol width to a smaller number and later scaling it up.

When resizing the symbol down, the module outputs the 'index' of the packet. This 'index' output can be used as the 'idx hint' (index hint) input of another module that will increase the width of the packet symbols back to the original. This is done to repair 'index' if the value has overflown due to low number of bits in the symbol while scaled down. When decreasing the width of symbols the 'idx hint' input should be a single bit constant equal to zero. When increasing the width of a symbol, the 'idx hint' should be an index of a packet that entered the system later than the current packet and the width of this input should equal to the requested width of the symbol at the output of such block.

Signal Processing Module: this module processes packets of data from the input bus and outputs them onto a output bus. For further documentation see 'comm_library.doc'.

Parameters

Output Signal Width: 12

Output Signal Type Width: 3

[✓] Display Packet Type Information

Accept Packets of Type: 1

Generate Packets of Type: 1

Iteration Addition: 0

[✓] Display Signal Parameters

Signal Width: 12

Signal Binary Point: 0

Signal Is Signed: 0

Packet Type Width: 3

Figure B.14: Module's mask (parameter dialog)

Table B.2 Module parameters

Processing Parameters Internal variable name	The following 3 parameters define how packets will be processed by this module.
Accept Packets of Type p_AcceptPacketType	All modules selectively accept only packets of certain type. Only packets that have the same type as indicated by this parameter will be processed by this module.
Generate Packets of Type p_ProcessedPacketType	The packets on the output line will be generated with this *packet type*. This allows changing the type of a packet after being processed by a module.
Iteration Addition p_IterationAddition	This number is added to the *iteration* parameter in the packet header when the packet is processed by the module.
Signal Parameters Internal variable name	The following 4 parameters specify what the module should assume about the input signal. If a parameter is greyed than either the module assumes it to be of a specific value (this value is listed in the greyed field) or the module does not care (a meaningless reference to a Matlab variable is displayed in the greyed field).
Signal Width p_SignalWidth	Bit width of the signal bus.
Signal Binary Point p_SignalBinaryPoint	Binary point if the signal symbol is interpreted as a number (if greyed, see above).
Signal Is Signed p_bSignalIsSigned	Signed / Unsigned if the signal symbol is interpreted as a number (if greyed, see above).
Signal Type Width p_SignalTypeWidth	The width of the *packet type* portion of the *header* symbol of the packet (see Packet Specifications).
Output Signal Parameters Internal variable name	The following 4 parameters specify the output signal parameters in modules that change the format of the line.
Signal Width p_OutputSignalWidth	Bit width of the signal bus.
Signal Binary Point p_OutputSignalBinaryPoint	Binary point if the signal symbol is interpreted as a number (if greyed, see above).
Signal Is Signed p_bOutputSignalIsSigned	Signed / Unsigned if the signal symbol is interpreted as a number (if greyed, see above).
Signal Type Width p_OutputSignalTypeWidth	The width of the *packet type* portion of the *header* symbol of the packet (see Packet Specifications).

B.3.1 Mask Customization

All modules should have the same outward look, thus certain rules apply to the creation of module masks:

- All modules **must be masked**. All parameters of the module must be brought to the mask. The modules, blocks and logic inside a module are not allowed to use *Matlab* variables; only parameters from the mask can be used. Additional variables can be created inside the mask using the initialization pane of the mask. For an example, see *Parallel FIR Filter* module. This module uses the command:
  ```
      [m_Rate,m_IRLength] = size(p_ImpulseResponse)
  ```
  This command computes the temporary variables `m_Rate` and `m_IRLength` to make the expressions used to customize the logic inside the module simpler.
- The mask must be similar to the masks of other modules. The mask of a common module allows hiding certain parameters. The checkboxes *Display Packet Type Information* and *Display Signal Parameters* allow you to hide the Packet Type Information and the Signal Parameters respectively.
  This is implemented using a call-back function *HidefieldsFcn*. Specific variable names inside the mask are used for the *HidefieldsFcn* to work. The checkbox's variable name has to begin with *p_ bView* and a second separator parameter with the variable name beginning with *p_ eView* has to be created to mark the end of the section of the parameters to hide (this is a dummy parameter that will be hidden). The function simply hides or shows all the parameters between the check box parameter and the separator. The function *SetCallbacks* sets the *HidefieldsFcn* as a call-back for all parameters with the variable name beginning with *p_ bView* (nonetheless, you can enter the call-backs manually).
- As the *icon* (the visual representation of a module in Simulink) of the module use the color transition bitmap. To edit the icon, open the mask editor and enter the drawing code into the *Icon* pane. This is the default drawing code (draw color transition background using a bmp image and label the ports appropriately) for a general block:

```
image(imread('decoder.bmp'));

port_label('output',2,'Line Out');
port_label('input',1,'Line In');
```

- *Variable names* of all mask parameters should begin with '*p_*'. Names of variables defined (for local use of the module) in the initialization section of the mask should begin with '*m_*'. A name of a variable that is of *boolean* type (true/false) should contain a '*b*' prefix after the '*p_*', e.g., '*p_ bIsItRaining*' or '*p_ bRaining*'.
- If a parameter must be of a certain value for a specific module (for example a module might work only on signed numbers) then the parameter should not be omitted, but the default value should be assigned to the parameter and the parameter should be greyed (disabled) in the mask.
- If a module does not use a value of a parameter (a permutation module does depend on whether its dealing with signed or unsigned numbers) the parameter should be greyed (disabled) and the default value assigned to the variable should be a meaningless reference to a default Matlab variable (e.g., 'bSignalIsSigned'). The following are the default Matlab variables for signal properties: *SignalWidth, SignalBinaryPoint, bSignalIsSigned, SignalTypeWidth* and the usage of these variables is obvious from the parameter's name.
- If a signed/unsigned dropdown box inside a design has to be updated according to a parameter from the mask of the module, create a call-back function and let this function be called when the block is updated. For an example, see *IsSignedFcnVectorPacketGenerator.* This function has to be entered in the *Initialization* pane of the *Mask Editor.* For the name of the function use: *IsSignedFcn<module_ name>.*

B.4 List of Support *Matlab* functions

In addition to *Matlab* scripts contained within modules, the functions listed in Table B.3 are used in the library.

Table B.3: Support *Matlab* functions

Function	Description
cmlGetGenerateID	Create unique identifier for the Generate block
CreateFSM	Create a Finite State Machine description structure from the Transition and Outputs matrices
CreateRandomFSM	Create a randomized state machine with the given dimensions
FirstStateOccurence	Used internally by the *Viterbi* module to optimize its performance
GenTailTree	Used by the *Viterbi* module to generate the tree of tail-sequences
IsSigned...	Callbacks used to set the parameters of blocks within modules
LineParse	Parse line data into a cell array of packets
StateTransitionCodes	Used in the Viterbi module to determine codes for storing the track-back information
ViewMem	Parse the output of the *MemorySampled ToWorkspace* block
Generate43Decoder	Create lookup table for SISO decoding of the (4,3) parity check code
GetRotationPermutation	Compute a permutation that will "rotate" a n-dimensional hypercube
pdec	Parity check code SISO decoder
tpc_channel	*Matlab* model of the channel used in the TPC system (AWGN)
tpc_dec	*Matlab* implementation of general parity check product code decoder
tpc_enc	*Matlab* implementation of general parity check product code encoder

Appendix C

Designs and Source Code

Please contact Martin.Irman@mail.mcgill.ca or Jan.Bajcsy@mcgill.ca for information on how to obtain a copy of the designs and source code of the modules used in this thesis.

References

[1] G. P. Agrawal, *Fiber-optic Communication Systems,* Wiley-Interscience, New York, NY, 2002.

[2] P. J. Ashenden, *The Student's Guide to VHDL*, Morgan Kaufman Publishers, San Francisco, CA, 1998.

[3] L. Bahl, J. Cocke, F. Jelinek, R. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Transactions on Inform. Theory*, vol. 20, pp. 284-287, March 1974.

[4] C. Berrou, A. Glavieux, P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," *Proceedings of the IEEE International Conference on Communications*, Geneva, Switzerland, pp. 1064-1070, May 1993.

[5] J. Bhasker, *A SystemC Primer*, Star Galaxy Publ., Allentown, PA, 2002.

[6] Jinian Bian, Hongxi Xue, Ming Su, "VIDE: a visual VHDL integrated design environment," *Proceedings of the Asia and South Pacific Design Automation Conference*, In, Japan, pp. 383-386, Jan. 1997.

[7] M. D. Ciletti, *Advanced Digital Design with the Verilog HDL*, PerformanceCAE Publishing, Mountain View, CA, 1997.

[8] Communication Research Centre Canada, *Ultra-fast FEC Codecs*, www.crc.ca/en/html/fec/home/codecs/codecs.

[9] M. R. Dale, and R. M. Gagliardi, "Channel coding for asynchronous fiber optic CDMA communications," *IEEE Transactions on Comm*unications, pp. 2485-2492, Sept. 1995.

[10] G. D. Forney, Jr., "Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference," *IEEE Transactions on Information Theory*, vol. 18, pp. 363-378, May 1972.

[11] A. A. Garba, M. Irman, J. Bajcsy, "Enabling real-time optical CDMA: Towards a fully customizable FPGA-based communication library," IEEE International Conference on Acoustics, Speech and Signal Processing – invited poster presentation and demonstration at Xilinx Corp. booth, Montreal, QC, May 2004.

[12] A. A. Garba, R. M. H. Yim, J. Bajcsy, L. Chen, "Analysis of optical CDMA signal transmission: capacity limits and simulation results," *EURASIP Journal on Applied Signal Processing*, vol. 2005, pp. 1603-1616, Spring 2005.

[13] K. Garcie, S. Crozier, A. Hunt, "Performance of a low-complexity Turbo decoder with a simple early stopping criterion implemented on a SHARC processor," *Proceedings of the International Mobile Satellite Conference*, Ottawa, ON, pp. 281-286, June 1999.

[14] W. J. Gross, P. G. Gulak, "Simplified MAP algorithm suitable for implementation of turbo decoders," *Electronics Letters*, vol. 34, pp. 1577-1578, Aug. 1998.

[15] J. Hagenauer, P. Hoeher, "A Viterbi algorithm with soft-decision outputs and its applications," *Proceedings of the IEEE Global Telecommunications Conference*, Dallas, TX, pp. 47.1.1-47.1.6, Nov. 1989.

[16] Intel Corp., *Intel Pentium 4 Processor*, http://www.intel.com/products/processor/pentium4HTXE/index.htm.

[17] M. Irman, J. Bajcsy, "Generation of regular logic cell structures for coding in a real-time optical CDMA network," *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, Saskatoon, SK, May 2005.

[18] M. Irman, J. Bajcsy, "A Rapid System Prototyping Platform for Error Control Coding in Optical CDMA Networks," *Proceedings of the IEEE International Workshop on Rapid Systems Prototyping*, Montreal, QC, pp. 232-234, June 2005.

[19] M. Irman, J. Bajcsy, "On computational intractability of path counting on a general directed graph," IEEE International Symposium on Information Theory, Recent Results Poster Session, Chicago, IL, June 2004.

[20] Mathworks Corp., *Matlab/Simulink, Release 13*, www.mathworks.com.

[21] Mentor Graphics Corp., *ModelSim*, www.model.com.

[22] D. L. Miller-Karlow, E. J. Golin, "vVHDL: a visual hardware description language," *Proceedings of the IEEE Workshop on Visual Languages*, Seattle, WA, pp. 133-139, Sept. 1992.

[23] Nallatech, *Virtex-II XtremeDSP Development Kit*, http://www.nallatech.com/?node_id=1.2.2&id=24.

[24] G. Park, S. Yoon, C. Kang, D. Hong, "An implementation method of a turbo-code decoder using block-wise MAP algorithm," *Proceedings of the Vehicular Technology Conference*, Boston, MA, vol. 6, pp. 2956-2961, Sept. 2000.

[25] L. Pinuel, I. Martin, F. Tirado, "A special-purpose parallel computer for solving partial differential equations," *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, Madrid, Spain, pp. 509-517, Jan. 1998.

[26] J. G. Proakis, *Digital Communications*, McGraw-Hill, New York, NY, 2001.

[27] P. R. Prucnal, M. A. Santoro, T. R. Fan, " Spread spectrum fiber-optic local area network," *IEEE Journal of Lightwave Technology*, vol. 4, pp. 547-554, May 1986.

[28] U. Ramacher, U. Rückert, *VLSI Design of Neural Networks*, Kluwer Academic Publishers, Boston, MA, 1991.

[29] J. A. Salehi, "Code division multiple-access techniques in optical fiber networks. I. Fundamental Principles," *IEEE Transactions on Communications*, vol. 37, pp. 824-833, Aug. 1989.

[30] J. A. Salehi, "Code division multiple-access techniques in optical fiber networks. II. System performance analysis," *IEEE Transactions on Communications*, vol. 37, pp. 834-842, Aug. 1989.

[31] G. Scragg, D. Baldwin, *Algorithms and Data Structures: The Science of Computing*, Charles River Media, Florence, KY, 2004.

[32] S. Swanchara, S. Harper, P. Athanas, "A stream-based configurable computing radio testbed," *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, pp. 40-47, Apr. 1998.

[33] Texas Instruments, *DSP Platforms*, http://dspvillage.ti.com.

[34] M. Vasilko, L. Machacek, M. Matej, P. Stepien, S. Holloway, "A rapid prototyping methodology and platform for seamless communication

systems," *IEEE International Workshop on Rapid System Prototyping*, Monterey, CA, pp. 70-76, June 2001.

[35] S. Verdu, *Multiuser Detection*, Cambridge University Press, Cambridge, UK, 1998.

[36] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, pp. 260-269, April 1967.

[37] S. B. Wickers, S. Kim, *Fundamentals of Codes, Graphs and Iterative Decoding,* Kluwer Academic Publishers, Boston, MA, 2003.

[38] Xilinx Corp., *Forge*, www.xilinx.com/ise/advanced/forge_get.htm.

[39] Xilinx Corp., *Forward Error Correction Solution*, www.xilinx.com/ipcenter/fec_index.htm.

[40] Xilinx Corp., *ISE, Version 6.3i*, www.xilinx.com/products/design_resources/design_tool/index.htm.

[41] Xilinx Corp., *Virtex-II Platform FPGA Handbook*, UG002 (v1.0), www.xilinx.com, San Jose, CA, Dec. 2000.

[42] Xilinx Corp., *Virtex-II Pro Platform FPGA Handbook*, UG012 (v1.0), www.xilinx.com, San Jose, CA, Jan. 2002.

[43] Xilinx Corp., *Xilinx System Generator for DSP, Version 6.3i*, www.xilinx.com.

[44] R. M. H. Yim, *New Approaches to Optical Code-Division Multiple Access*, Master's thesis, McGill University, Montreal, QC, July 2002.

[45] R. M. H. Yim, J. Bajcsy, L. R. Chen, "Optical CDMA transmission enabled by turbo codes," *Proc. International Symposium on Turbo Codes*, Brest, France, pp. 571-574, Sept. 2003.

[46] R. E. Ziemer, R. L. Peterson, *Introduction to Digital Communication*, Prentice Hall, Upper Saddle River, NJ, 2001.